mirdata

Release 0.3.8

The mirdata development team

Feb 02, 2024

CONTENTS

`mirdata` is an open-source Python library that provides tools for working with common Music Information Retrieval (MIR) datasets, including tools for:

- downloading datasets to a common location and format
- validating that the files for a dataset are all present
- loading annotation files to a common format, consistent with `mir_eval`
- parsing track level metadata for detailed evaluations.

```
pip install mirdata
```

For more details on how to use the library see the *Tutorial*.

CHAPTER

ONE

CITING MIRDATA

If you are using the library for your work, please cite the version you used as indexed at Zenodo:

If you refer to mirdata's design principles, motivation etc., please cite the following paper[1]:

When working with datasets, please cite the version of `mirdata` that you are using (given by the `DOI` above) **AND** include the reference of the dataset, which can be found in the respective dataset loader using the `cite()` method.

[1] Rachel M. Bittner, Magdalena Fuentes, David Rubinstein, Andreas Jansson, Keunwoo Choi, and Thor Kell. "mirdata: Software for Reproducible Usage of Datasets." In Proceedings of the 20th International Society for Music Information Retrieval (ISMIR) Conference, 2019.:

CHAPTER

TWO

CONTRIBUTING TO MIRDATA

We welcome contributions to this library, especially new datasets. Please see *Contributing* for guidelines.

- Issue Tracker
- Source Code

2.1 Overview

```
pip install mirdata
```

`mirdata` is a library which aims to standardize how audio datasets are accessed in Python, removing the need for writing custom loaders in every project, and improving reproducibility. Working with datasets usually requires an often cumbersome step of downloading data and writing load functions that load related files (for example, audio and annotations) into a standard format to be used for experimenting or evaluating. `mirdata` does all of this for you:

```
import mirdata

print(mirdata.list_datasets())

tinysol = mirdata.initialize('tinysol')
tinysol.download()

# get annotations and audio for a random track
example_track = tinysol.choice_track()
instrument = example_track.instrument_full
pitch = example_track.pitch
y, sr = example_track.audio
```

`mirdata` loaders contain methods to:

- `download()`: download (or give instructions to download) a dataset
- `load_*()`: load a dataset's files (audio, metadata, annotations, etc.) into standard formats, so you don't have to write them yourself which are compatible with `mir_eval` and `jams`.
- `validate()`: validate that a dataset is complete and correct
- `cite()`: quickly print a dataset's relevant citation
- access `track` and `multitrack` objects for grouping multiple annotations for a particular track/multitrack
- and more

See the *Tutorial* for a detailed explanation of how to get started using this library.

2.1.1 mirdata design principles

Ease of use and contribution

We designed `mirdata` to be easy to use and easy to contribute to. `mirdata` simplifies the research pipeline considerably, facilitating research in a wider diversity of tasks and musical datasets. We provide detailed examples on how to interact with the library in the *Tutorial*, as well as detail explanation on how to contribute in *Contributing*. Additionally, we have a repository of Jupyter notebooks with usage examples of the different datasets.

Reproducibility

We aim for `mirdata` to aid in increasing research reproducibility by providing a common framework for MIR researchers to compare and validate their data. If mistakes are found in annotations or audio versions change, using `mirdata`, the community can fix mistakes while still being able to compare methods moving forward.

canonical versions

The `dataset loaders` in `mirdata` are written for what we call the `canonical version` of a dataset. Whenever possible, this should be the official release of the dataset as published by the dataset creator/s. When this is not possible, (e.g. for data that is no longer available), the procedure we follow is to find as many copies of the data as possible from different researchers (at least 4), and use the most common one. To make this process transparent, when there are doubts about the data consistency we open an issue and leave it to the community to discuss what to use.

Standardization

Different datasets have different annotations, metadata, etc. We try to respect the idiosyncrasies of each dataset as much as we can. For this reason, `tracks` in each `Dataset` in `mirdata` have different attributes, e.g. some may have `artist` information and some may not. However there are some elements that are common in most datasets, and in these cases we standardize them to increase the usability of the library. Some examples of this are the annotations in `mirdata`, e.g. `BeatData`.

2.1.2 indexes

Indexes in *mirdata* are manifests of the files in a dataset and their corresponding md5 checksums. Specifically, an index is a json file with the mandatory top-level key `version` and at least one of the optional top-level keys `metadata`, `tracks`, `multitracks` or `records`. An index might look like:

Example Index

```
{   "version": "1.0.0",
    "metadata": {
        "metadata_file_1": [
                // the relative path for metadata_file_1
                "path_to_metadata/metadata_file_1.csv",
                // metadata_file_1 md5 checksum
                "bb8b0ca866fc2423edde01325d6e34f7"
            ],
        "metadata_file_2": [
                // the relative path for metadata_file_2
```

(continues on next page)

(continued from previous page)

```
                "path_to_metadata/metadata_file_2.csv",
                // metadata_file_2 md5 checksum
                "6cce186ce77a06541cdb9f0a671afb46"
            ]
        }
    "tracks": {
        "track1": {
            'audio': ["audio_files/track1.wav", "6c77777ce77a06541cdb9f0a671afb46"],
            'beats': ["annotations/track1.beats.csv", "ab8b0ca866fc2423edde01325d6e34f7
↪"],
            'sections': ["annotations/track1.sections.txt",
↪"05abeca866fc2423edde01325d6e34f7"],
        }
        "track2": {
            'audio': ["audio_files/track2.wav", "6c77777ce77a06542cdb9f0a672afb46"],
            'beats': ["annotations/track2.beats.csv", "ab8b0ca866fc2423edde02325d6e34f7
↪"],
            'sections': ["annotations/track2.sections.txt",
↪"05abeca866fc2423edde02325d6e34f7"],
        }
        ...
        }
}
```

The optional top-level keys (*tracks*, *multitracks* and *records*) relate to different organizations of music datasets. *tracks* are used when a dataset is organized as a collection of individual tracks, namely mono or multi-channel audio, spectrograms only, and their respective annotations. *multitracks* are used in when a dataset comprises of multitracks - different groups of tracks which are directly related to each other. Finally, *records* are used when a dataset consists of groups of tables (e.g. relational databases), as many recommendation datasets do.

See the contributing docs *1. Create an index* for more information about mirdata indexes.

2.1.3 annotations

mirdata provides `Annotation` classes of various kinds which provide a standard interface to different annotation formats. These classes are compatible with the `mir_eval` library's expected format, as well as with the jams format. The format can be easily extended to other formats, if requested.

2.1.4 metadata

When available, we provide extensive and easy-to-access `metadata` to facilitate track metadata-specific analysis. `metadata` is available as attributes at the `track` level, e.g. `track.artist`.

2.2 Supported Datasets and Annotations

2.2.1 Dataset Quick Reference

This table is provided as a guide for users to select appropriate datasets. The list of annotations omits some metadata for brevity, and we document the dataset's primary annotations only. The number of tracks indicates the number of unique "tracks" in a dataset, but it may not reflect the actual size or diversity of a dataset, as tracks can vary greatly in length (from a few seconds to a few minutes), and may be homogeneous.

"Downloadable" possible values:

- : Freely downloadable
- : Available upon request
- : Do-it-yourself download
- : Features only
- : Not available

Find the API documentation for each of the below datasets in *Initializing*.

Dataset	Downloadable?	Annotation Types	Tracks	License
AcousticBrainz Genre	• audio: • annotations:	• *Genre*	>4M	•  • Custom
BAF	• audio: • annotations:	• *Matches*	3425	•
Ballroom	• audio: • annotations:	• *Beats* • *Tempo*	698	
Beatles	• audio: • annotations:	• *Beats* • *Chords* • *Sections* • *Key* • *Vocal Activity*	180	
Beatport EDM key	• audio: • annotations:	• global *Key*	1486	

continues on next page

Table 1 – continued from previous page

Dataset	Downloadable?	Annotation Types	Tracks	License
Billboard (McGill)	• audio: • annotations:	• *Chords* • *Sections*	890	CC0
Candombe	• audio: • annotations:	• *Beats*	35	CC BY-NC-SA
cante100	• audio: • annotations:	• *F0* • Vocal *Notes*	100	Custom
CIPI	• musicXML: • embeddings: • annotations:	• difficulty levels	652	CC BY-NC-SA
(CompMusic) Carnatic Rhythm	• audio: • annotations:	• *Beats* • *Meter*	176	CC BY-NC-SA
(CompMusic) Hindustani Rhythm	• audio: • annotations:	• *Beats* • *Meter*	151	CC BY-NC-SA
(CompMusic) Indian Tonic	• audio: • annotations:	• *Tonic*	2150	CC BY-NC-SA
(CompMusic) Jingju A Cappella	• audio: • annotations:	• *Lyrics* • *Phonemes* • *Syllables*	82	CC BY-NC-SA
(CompMusic) OTMM Makam	• audio: • annotations:	• *F0* • *Tonic*	1000	CC BY-NC-SA
(CompMusic) Raga	• audio: • annotations:	• *F0* • *Segments* • *Tonic*	780	CC BY-NC-SA

continues on next page

Table 1 – continued from previous page

Dataset	Downloadable?	Annotation Types	Tracks	License
Dagstuhl ChoirSet	• multitrack audio: • annotations:	• *F0* • *Beats* • *Notes*	108	CC BY
DALI	• audio: • annotations:	• *Lyrics* • Vocal *Notes*	5358	CC BY-SA
Da-TACOS	• audio: • annotations:	• *Lyrics* • Vocal *Notes*	• benchmark 15000 tracks • coveranalysis 10000 tracks	CC BY-SA
EGFxSet	• audio: • annotations:	• *Notes*	8970	CC BY-SA
Filosax	• audio: • annotations: • midi:	• *F0* • *Beats* • *Chords* • *Tempo* • *Notes*	48	
Four-Way Tabla Stroke	• audio: • annotations:	• *Tags*	236	CC BY-SA
Freesound One-Shot Percussive Sounds	• audio: • annotations:	• *Tags*	10254	CC BY-SA
Giantsteps key	• audio: • annotations:	global *Key*	500	CC BY-SA
Giantsteps tempo	• audio: • annotations:	• global *Genre* • global *Tempo*	664	CC BY-SA

continues on next page

Table 1 – continued from previous page

Dataset	Downloadable?	Annotation Types	Tracks	License
Good Sounds	• audio: : • annotations:	• instrument *Instruments* • sound quality metadata • instrument metadata	16308	CC BY-SA
Groove MIDI	• audio: • midi:	• *Beats* • *Tempo* • *Drums*	1150	CC BY-SA
Gtzan-Genre	• audio: • annotations:	• global *Genre* • *Beats* • *Tempo*	1000	
Guitarset	• audio: • midi:	• *Beats* • *Chords* • *Key* • *Tempo* • *Notes* • *F0*	360	
Ikala	• audio: • annotations:	• Vocal *F0* • *Lyrics*	252	Custom
Haydn op20	• audio: N/A • midi: • scores: • annotations:	• symbolic *Chords* • symbolic *Key*	24	CC BY-NC-SA
IDMT-SMT-Audio Effects	• audio: • annotations:	• instrument *Instruments* • midi nr *Notes* • metadata *Effect*	55044	CC BY-NC-ND
IRMAS	• audio: • annotations:	• *Instruments* • *Genre*	9579	CC BY-NC-SA

continues on next page

Table 1 – continued from previous page

Dataset	Downloadable?	Annotation Types	Tracks	License
MTG_jamendo_autot	• audio: • annotations:	• moodtheme annotations	18448	(cc) BY-NC-SA
MAESTRO	• audio: • annotations:	Piano *Notes*	1282	(cc) BY-NC-SA
Medley-solos-DB	• audio: • annotations:	*Instruments*	21571	(cc) BY-SA
MedleyDB melody	• audio: • annotations:	Melody *F0*	108	(cc) BY-NC-SA
MedleyDB pitch	• audio: • annotations:	• *F0* • *Instruments*	103	(cc) BY-NC-SA
Mridangam Stroke	• audio: • annotations:	• *Stroke Name* • *Tonic*	6977	(cc) BY
Orchset	• audio: • annotations:	Melody *F0*	64	(cc) BY-NC-SA
PHENICX-Anechoic	• multitrack audio: • annotations:	• Aligned score *Notes* • Original score *Notes*	4	(cc) BY-NC-SA
Queen	• audio: • annotations:	• *Chords* • *Sections* • *Key*	51	
RWC classical	• audio: • annotations:	• *Beats* • *Sections*	61	Custom
RWC jazz	• audio: • annotations:	• *Beats* • *Sections*	50	Custom

continues on next page

Table 1 – continued from previous page

Dataset	Downloadable?	Annotation Types	Tracks	License
RWC popular	• audio: • annotations:	• *Beats* • *Sections* • *Vocal Activity* • *Chords* • *Tempo*	100	Custom
Salami	• audio: • annotations:	*Sections*	1359	CC ZERO
Saraga Carnatic	• audio: • annotations:	• *F0* • Vocal *F0* • *Tempo* • *Phrases* • *Beats* (samas) • *Sections* • *Tonic*	249	CC BY-NC-SA
Saraga Hindustani	• audio: • annotations:	• *F0* • *Tempo* • *Phrases* • *Beats* (samas) • *Sections* • *Tonic*	108	CC BY-NC-SA
Saraga-Carnatic-Melody-Synth (SCMS)	• audio: • annotations:	• *F0* • *Events*	2460	CC BY-NC-SA
Slakh	• multitrack audio: • annotations:	• Notes *Notes* • Instruments *Instruments*	1710	CC BY
Tinysol	• audio: • annotations:	• *Instruments* • *Technique* • *Notes*	2913	CC BY
Tonality ClassicalDB	• audio: • annotations:	Global *Key*	881	CC BY-NC-SA

continues on next page

Table 1 – continued from previous page

Dataset	Downloadable?	Annotation Types	Tracks	License
TONAS	• audio: • annotations:	• *F0* • *Notes*	72	Custom
vocadito	• audio: • annotations:	• *F0* • *Notes* • *Lyrics*	40	

2.2.2 Annotation Types

The table above provides annotation types as a guide for choosing appropriate datasets, but it is difficult to generically categorize annotation types, as they depend on varying definitions and their meaning can change depending on the type of music they correspond to. Here we provide a rough guide to the types in this table, but we **strongly recommend** reading the dataset specific documentation to ensure the data is as you expect. To see how these annotation types are implemented in `mirdata` see *Annotations*.

Events

A generic annotation to indicate whether a particular event is happening at a given time. It can be used, for instance, to indicate whether a particular instrument is playing at a given time-step or whether a particular note is being played at a given time-step. In fact, it is implicit in annotations such as *F0* or Vocal *Notes* (instrument is activated when the melody is non-0). However, some datasets provide it as a standalone event annotation.

Beats

Musical beats, typically encoded as sequence of timestamps and corresponding beat positions. This implicitly includes *downbeat* information (the beginning of a musical measure).

Chords

Musical chords, e.g. as might be played on a guitar. Typically encoded as a sequence of labeled events, where each event has a start time, end time, and a label. The label taxonomy varies per dataset, but typically encode a chord's root and its quality, e.g. A:m7 for "A minor 7".

Drums

Transcription of the drums, typically encoded as a sequence of labeled events, where the labels indicate which drum instrument (e.g. cymbal, snare drum) is played. These events often overlap with one another, as multiple drums can be played at the same time.

F0

Musical pitch contours, typically encoded as time series indicating the musical pitch over time. The time series typically have evenly spaced timestamps, each with a corresponding pitch value which may be encoded in a number of formats/granularities, including midi note numbers and Hertz.

Effect

Effect applied to a track. It may refer to the effect applied to a single stroke or an entire track. It can include the effect name, the effect type, the effect parameters, and the effect settings.

Genre

A typically global "tag", indicating the genre of a recording. Note that the concept of genre is highly subjective and we refer those new to this task to this article.

Instruments

Labels indicating which instrument is present in a musical recording. This may refer to recordings of solo instruments, or to recordings with multiple instruments. The labels may be global to a recording, or they may vary over time, indicating the presence/absence of a particular instrument as a time series.

Key

Musical key. This can be defined globally for an audio file or as a sequence of events.

Lyrics

Lyrics corresponding to the singing voice of the audio. These may be raw text with no time information, or they may be time-aligned events. They may have varying levels of granularity (paragraph, line, word, phoneme, character) depending on the dataset.

Matches

Music identifications in a query audio. This term is used in Audio Fingerprinting to refer to identifications of music from a reference database. Matches include information about which reference audio has been identified and the start and end times of the query match.

Meter

Rhythmic meter for each measure. A classical example of meter in Western music would be 4/4. Details how many subdivisions and the length of this subdivisions that we do have per each measure.

Melody

The musical melody of a song. Melody has no universal definition and is typically defined per dataset. It is typically encoded as *F0* or as *Notes*. Other types of annotations such as Vocal F0 or Vocal Notes can often be considered as melody annotations as well.

Notes

Musical note events, typically encoded as sequences of start time, end time, label. The label typically indicates a musical pitch, which may be in a number of formats/granularities, including midi note numbers, Hertz, or pitch class.

Phonemes

Sung phonemes of the lead vocal lyrics. Likewise the annotations of lyrics, it can be represented as a stream of characters, or it can be time-aligned by start and end times, and the phoneme comprised in each interval.

Phrases

Musical phrase events, typically encoded by a sequence of timestamps indicating the boundary times and defined by solfège symbols. This annotations are not intended to describe the complete melody but the musical phrases present in the track.

Sections

Musical sections, which may be "flat" or "hierarchical", typically encoded by a sequence of timestamps indicating musical section boundary times. Section annotations sometimes also include labels for sections, which may indicate repetitions and/or the section type (e.g. Chorus, Verse).

Segments

Segments of particular musical events, e.g. segments of note stability, segments of particular melodic event, and many more.

Technique

The playing technique used by a particular instrument, for example "Pizzicato". This label may be global for a given recording or encoded as a sequence of labeled events.

Tempo

The tempo of a song, typical in units of beats-per-minute (bpm). This is often indicated globally per track, but in practice tracks may have tempos that change, and some datasets encode tempo as time-varying quantity. Additionally, there may be multiple reasonable tempos at any given time (for example, often 2x or 0.5x a tempo value will also be "correct"). For this reason, some datasets provide two or more different tempo values.

Vocal Activity

A time series or sequence of events indicating when singing voice is present in a recording. This type of annotation is implicitly available when Vocal *F0* or Vocal *Notes* annotations are available.

Stroke Name

An open "tag" to identify an instrument stroke name or type. Used for instruments that have specific stroke labels.

Syllables

Additional representation of the sung lyrics but structured as syllables instead of complete sentences. It can be annotated as time-aligned events where the events are the syllables happening at certain time intervals. Otherwise, they can be represented as a stream of strings, grouped by meaningful syllable structures.

Tags

This is a broad annotation type that is used to label music and sounds, that often spans multiple categories. For example, music can be labeled with tags pertaining to the instruments present, the musical style, the mood, etc. Tags are often free-form and may not have a structured taxonomy/set of labels. They are typically represented as a list of strings, sometimes with associated weights/confidences.

Tonic

The absolute tonic of a track. It may refer to the tonic a single stroke, or the tonal center of a track.

2.3 Tutorial

2.3.1 Installation

To install `mirdata`:

```
pip install mirdata
```

2.3.2 Usage

`mirdata` is easily imported into your Python code by:

```
import mirdata
```

Initializing a dataset

Print a list of all available dataset loaders by calling:

```
import mirdata
print(mirdata.list_datasets())
```

To use a loader, (for example, 'orchset') you need to initialize it by calling:

```
import mirdata
orchset = mirdata.initialize('orchset')
```

Now `orchset` is a `Dataset` object containing common methods, described below.

Downloading a dataset

All dataset loaders in `mirdata` have a `download()` function that allows the user to download the canonical version of the dataset (when available). When initializing a dataset, by default, mirdata will download/read data to/from a default location ("~/mir_datasets"). This can be customized by specifying *data_home* in *mirdata.initialize*.

Downloading a dataset into the default folder:

In this first example, `data_home` is not specified. Thus, ORCHSET will be downloaded and retrieved from `mir_datasets` folder created at user root folder:

```
import mirdata
orchset = mirdata.initialize('orchset')
orchset.download()  # Dataset is downloaded to ~/mir_datasets/orchset
```

Downloading a dataset into a specified folder:

Now `data_home` is specified and so orchset will be read from / written to this custom location:

```
orchset = mirdata.initialize('orchset', data_home='Users/leslieknope/Desktop/
↪orchset123')
orchset.download()  # Dataset is downloaded to the folder "orchset123" Leslie Knope
↪'s desktop
```

Partially downloading a dataset

The `download()` functions allows partial downloads of a dataset. In other words, if applicable, the user can select which elements of the dataset they want to download. Each dataset has a `REMOTES` dictionary were all the available elements are listed.

`cante100` has different elements as seen in the `REMOTES` dictionary. Thus, we can specify which of these elements are downloaded, by passing to the `download()` function the list of keys in `REMOTES` that we are interested in. This list is passed to the `download()` function through the `partial_download` variable.

Example REMOTES

```
REMOTES = {
    "spectrogram": download_utils.RemoteFileMetadata(
        filename="cante100_spectrum.zip",
        url="https://zenodo.org/record/1322542/files/cante100_spectrum.zip?download=1",
        checksum="0b81fe0fd7ab2c1adc1ad789edb12981",  # the md5 checksum
```

(continues on next page)

(continued from previous page)

```
        destination_dir="cante100_spectrum",  # relative path for where to unzip the␣
↪data, or None
    ),
    "melody": download_utils.RemoteFileMetadata(
        filename="cante100midi_f0.zip",
        url="https://zenodo.org/record/1322542/files/cante100midi_f0.zip?download=1",
        checksum="cce543b5125eda5a984347b55fdcd5e8",  # the md5 checksum
        destination_dir="cante100midi_f0",  # relative path for where to unzip the data,␣
↪or None
    ),
    "notes": download_utils.RemoteFileMetadata(
        filename="cante100_automaticTranscription.zip",
        url="https://zenodo.org/record/1322542/files/cante100_automaticTranscription.zip?
↪download=1",
        checksum="47fea64c744f9fe678ae5642a8f0ee8e",  # the md5 checksum
        destination_dir="cante100_automaticTranscription",  # relative path for where to␣
↪unzip the data, or None
    ),
    "metadata": download_utils.RemoteFileMetadata(
        filename="cante100Meta.xml",
        url="https://zenodo.org/record/1322542/files/cante100Meta.xml?download=1",
        checksum="6cce186ce77a06541cdb9f0a671afb46",  # the md5 checksum
    ),
    "README": download_utils.RemoteFileMetadata(
        filename="cante100_README.txt",
        url="https://zenodo.org/record/1322542/files/cante100_README.txt?download=1",
        checksum="184209b7e7d816fa603f0c7f481c0aae",  # the md5 checksum
    ),
}
```

A partial download example for `cante100` dataset could be:

```
cante100.download(partial_download=['spectrogram', 'melody', 'metadata'])
```

Validating a dataset

Using the method `validate()` we can check if the files in the local version are the same than the available canonical version, and the files were downloaded correctly (none of them are corrupted).

For big datasets: In future `mirdata` versions, a random validation will be included. This improvement will reduce validation time for very big datasets.

Accessing annotations

We can choose a random track from a dataset with the `choice_track()` method.

Loading annotations

```
random_track = orchset.choice_track()
print(random_track)
>>> Track(
       alternating_melody=True,
       audio_path_mono="user/mir_datasets/orchset/audio/mono/Beethoven-S3-I-ex1.wav",
       audio_path_stereo="user/mir_datasets/orchset/audio/stereo/Beethoven-S3-I-ex1.wav",
       composer="Beethoven",
       contains_brass=False,
       contains_strings=True,
       contains_winds=True,
       excerpt="1",
       melody_path="user/mir_datasets/orchset/GT/Beethoven-S3-I-ex1.mel",
       only_brass=False,
       only_strings=False,
       only_winds=False,
       predominant_melodic_instruments=['strings', 'winds'],
       track_id="Beethoven-S3-I-ex1",
       work="S3-I",
       audio_mono: (np.ndarray, float),
       audio_stereo: (np.ndarray, float),
       melody: F0Data,
   )
```

We can also access specific tracks by id. The available track ids can be accessed via the `.track_ids` attribute. In the next example we take the first track id, and then we retrieve the melody annotation.

```
orchset_ids = orchset.track_ids  # the list of orchset's track ids
orchset_data = orchset.load_tracks()  # Load all tracks in the dataset
example_track = orchset_data[orchset_ids[0]]  # Get the first track

# Accessing the track's melody annotation
example_melody = example_track.melody
```

Alternatively, we don't need to load the whole dataset to get a single track.

```
orchset_ids = orchset.track_ids  # the list of orchset's track ids
example_track = orchset.track(orchset_ids[0])  # load this particular track
example_melody = example_track.melody  # Get the melody from first track
```

Accessing data on non-local filesystems

mirdata uses the smart_open library, which supports non-local filesystems such as GCS and AWS. If your data lives, e.g. on Google Cloud Storage (GCS), simply set the `data_home` variable accordingly when initializing a dataset. For example:

Accessing annotations remotely

```
import mirdata

orchset = mirdata.initialize("orchset", data_home="gs://my-bucket/my-subfolder/orchset")

# everything should work the same as if the data were local
orchset.validate()

example_track = orchset.choice_track()
melody = example_track.melody
y, fs = example_track.audio_mono
```

Note that the data on the remote file system must have identical folder structure to what is specified by `dataset.download()`, and we do not support downloading (i.e. writing) to remote filesystems, only reading from them. To prepare a new dataset to use with mirdata, we recommend running `dataset.download()` on a local filesystem, and then manually transfering the folder contents to the remote filesystem.

mp3 data

For a variety of reasons, mirdata doesn't support remote reading of mp3 files, so some datasets with mp3 audio may have tracks unavailable attributes.

Annotation classes

`mirdata` defines annotation-specific data classes. These data classes are meant to standardize the format for all loaders, and are compatibly with jams and mir_eval.

The list and descriptions of available annotation classes can be found in *Annotations*.

Note: These classes may be extended in the case that a loader requires it.

Iterating over datasets and annotations

In general, most datasets are a collection of tracks, and in most cases each track has an audio file along with annotations.

With the `load_tracks()` method, all tracks are loaded as a dictionary with the ids as keys and track objects (which include their respective audio and annotations, which are lazy-loaded on access) as values.

```
orchset = mirdata.initialize('orchset')
for key, track in orchset.load_tracks().items():
    print(key, track.audio_path)
```

Alternatively, we can loop over the `track_ids` list to directly access each track in the dataset.

```
orchset = mirdata.initialize('orchset')
for track_id in orchset.track_ids:

    print(track_id, orchset.track(track_id).audio_path)
```

Basic example: including mirdata in your pipeline

If we wanted to use `orchset` to evaluate the performance of a melody extraction algorithm (in our case, `very_bad_melody_extractor`), and then split the scores based on the metadata, we could do the following:

mirdata usage example

```
import mir_eval
import mirdata
import numpy as np
import sox

def very_bad_melody_extractor(audio_path):
    duration = sox.file_info.duration(audio_path)
    time_stamps = np.arange(0, duration, 0.01)
    melody_f0 = np.random.uniform(low=80.0, high=800.0, size=time_stamps.shape)
    return time_stamps, melody_f0

# Evaluate on the full dataset
orchset = mirdata.initialize("orchset")
orchset_scores = {}
orchset_data = orchset.load_tracks()
for track_id, track_data in orchset_data.items():
    est_times, est_freqs = very_bad_melody_extractor(track_data.audio_path_mono)

    ref_melody_data = track_data.melody
    ref_times = ref_melody_data.times
    ref_freqs = ref_melody_data.frequencies

    score = mir_eval.melody.evaluate(ref_times, ref_freqs, est_times, est_freqs)
    orchset_scores[track_id] = score

# Split the results by composer and by instrumentation
composer_scores = {}
strings_no_strings_scores = {True: {}, False: {}}
for track_id, track_data in orchset_data.items():
    if track_data.composer not in composer_scores.keys():
        composer_scores[track_data.composer] = {}

    composer_scores[track_data.composer][track_id] = orchset_scores[track_id]
    strings_no_strings_scores[track_data.contains_strings][track_id] = \
        orchset_scores[track_id]
```

This is the result of the example above.

Example result

```
print(strings_no_strings_scores)
>>> {True: {
        'Beethoven-S3-I-ex1':OrderedDict([
               ('Voicing Recall', 1.0),
               ('Voicing False Alarm', 1.0),
               ('Raw Pitch Accuracy', 0.029798422436459245),
               ('Raw Chroma Accuracy', 0.08063102541630149),
               ('Overall Accuracy', 0.0272654370489174)
               ]),
        'Beethoven-S3-I-ex2': OrderedDict([
               ('Voicing Recall', 1.0),
               ('Voicing False Alarm', 1.0),
               ('Raw Pitch Accuracy', 0.009221311475409836),
               ('Raw Chroma Accuracy', 0.07377049180327869),
               ('Overall Accuracy', 0.008754863813229572)]),
        ...

        'Wagner-Tannhauser-Act2-ex2': OrderedDict([
               ('Voicing Recall', 1.0),
               ('Voicing False Alarm', 1.0),
               ('Raw Pitch Accuracy', 0.03685636856368564),
               ('Raw Chroma Accuracy', 0.08997289972899729),
               ('Overall Accuracy', 0.036657681940700806)])
        }}
```

You can see that `very_bad_melody_extractor` performs very badly!

Using mirdata with tensorflow

The following is a simple example of a generator that can be used to create a tensorflow Dataset.

mirdata with tf.data.Dataset example

```
import mirdata
import numpy as np
import tensorflow as tf

def orchset_generator():
    # using the default data_home
    orchset = mirdata.initialize("orchset")
    track_ids = orchset.track_ids
    for track_id in track_ids:
        track = orchset.track(track_id)
        audio_signal, sample_rate = track.audio_mono
        yield {
            "audio": audio_signal.astype(np.float32),
            "sample_rate": sample_rate,
            "annotation": {
```

(continues on next page)

(continued from previous page)

```
                "times": track.melody.times.astype(np.float32),
                "freqs": track.melody.frequencies.astype(np.float32),
            },
            "metadata": {"track_id": track.track_id}
        }

dataset = tf.data.Dataset.from_generator(
    orchset_generator,
    {
        "audio": tf.float32,
        "sample_rate": tf.float32,
        "annotation": {"times": tf.float32, "freqs": tf.float32},
        "metadata": {'track_id': tf.string}
    }
)
```

In future `mirdata` versions, generators for Tensorflow and Pytorch will be included.

2.4 Initializing

`mirdata.`**`initialize`**(*dataset_name*, *data_home=None*, *version='default'*)

Load a mirdata dataset by name

Example

```
orchset = mirdata.initialize('orchset')  # get the orchset dataset
orchset.download()  # download orchset
orchset.validate()  # validate orchset
track = orchset.choice_track()  # load a random track
print(track)  # see what data a track contains
orchset.track_ids()  # load all track ids
```

Parameters

- **dataset_name** (*str*) – the dataset's name see mirdata.DATASETS for a complete list of possibilities
- **data_home** (*str or None*) – path where the data lives. If None uses the default location.
- **version** (*str or None*) – which version of the dataset to load. If None, the default version is loaded.

Returns

Dataset – a mirdata.core.Dataset object

`mirdata.`**`list_datasets`**()

Get a list of all mirdata dataset names

Returns

list – list of dataset names as strings

2.5 Dataset Loaders

2.5.1 acousticbrainz_genre

Acoustic Brainz Genre dataset

Dataset Info

The AcousticBrainz Genre Dataset consists of four datasets of genre annotations and music features extracted from audio suited for evaluation of hierarchical multi-label genre classification systems.

Description about the music features can be found here: https://essentia.upf.edu/streaming_extractor_music.html

The datasets are used within the MediaEval AcousticBrainz Genre Task. The task is focused on content-based music genre recognition using genre annotations from multiple sources and large-scale music features data available in the AcousticBrainz database. The goal of our task is to explore how the same music pieces can be annotated differently by different communities following different genre taxonomies, and how this should be addressed by content-based genre r ecognition systems.

We provide four datasets containing genre and subgenre annotations extracted from four different online metadata sources:

- AllMusic and Discogs are based on editorial metadata databases maintained by music experts and enthusiasts. These sources contain explicit genre/subgenre annotations of music releases (albums) following a predefined genre namespace and taxonomy. We propagated release-level annotations to recordings (tracks) in AcousticBrainz to build the datasets.
- Lastfm and Tagtraum are based on collaborative music tagging platforms with large amounts of genre labels provided by their users for music recordings (tracks). We have automatically inferred a genre/subgenre taxonomy and annotations from these labels.

For details on format and contents, please refer to the data webpage.

Note, that the AllMusic ground-truth annotations are distributed separately at https://zenodo.org/record/2554044.

If you use the MediaEval AcousticBrainz Genre dataset or part of it, please cite our ISMIR 2019 overview paper:

```
Bogdanov, D., Porter A., Schreiber H., Urbano J., & Oramas S. (2019).
The AcousticBrainz Genre Dataset: Multi-Source, Multi-Level, Multi-Label, and Large-
↪Scale.
20th International Society for Music Information Retrieval Conference (ISMIR 2019).
```

This work is partially supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No 688382 AudioCommons.

class mirdata.datasets.acousticbrainz_genre.**Dataset**(*data_home=None*, *version='default'*)

The acousticbrainz genre dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –

- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

filter_index(*search_key*)

Load from AcousticBrainz genre dataset the indexes that match with search_key.

Parameters
search_key (*str*) – regex to match with folds, mbid or genres

Returns
dict – {*track_id*: track data}

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

get_random_track_splits(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

get_track_splits()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- **AttributeError** – If this dataset does not have tracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

license()

Print the license

load_all_train()

Load from AcousticBrainz genre dataset the tracks that are used for training across the four different datasets.

Returns
dict – {*track_id*: track data}

load_all_validation()

Load from AcousticBrainz genre dataset the tracks that are used for validating across the four different datasets.

Returns
dict – {*track_id*: track data}

`load_allmusic_train()`

Load from AcousticBrainz genre dataset the tracks that are used for validation in allmusic dataset.

Returns
dict – {*track_id*: track data}

`load_allmusic_validation()`

Load from AcousticBrainz genre dataset the tracks that are used for validation in allmusic dataset.

Returns
dict – {*track_id*: track data}

`load_discogs_train()`

Load from AcousticBrainz genre dataset the tracks that are used for training in discogs dataset.

Returns
dict – {*track_id*: track data}

`load_discogs_validation()`

Load from AcousticBrainz genre dataset the tracks that are used for validation in tagtraum dataset.

Returns
dict – {*track_id*: track data}

`load_extractor(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.acousticbrainz_genre.load_extractor

`load_lastfm_train()`

Load from AcousticBrainz genre dataset the tracks that are used for training in lastfm dataset.

Returns
dict – {*track_id*: track data}

`load_lastfm_validation()`

Load from AcousticBrainz genre dataset the tracks that are used for validation in lastfm dataset.

Returns
dict – {*track_id*: track data}

`load_multitracks()`

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

`load_tagtraum_train()`

Load from AcousticBrainz genre dataset the tracks that are used for training in tagtraum dataset.

Returns
dict – {*track_id*: track data}

`load_tagtraum_validation()`

Load from AcousticBrainz genre dataset the tracks that are used for validating in tagtraum dataset.

Returns
dict – {*track_id*: track data}

`load_tracks()`

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns
list – A list of track ids

`track_ids`

Return track ids

Returns
list – A list of track ids

`validate(verbose=True)`

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

`class mirdata.datasets.acousticbrainz_genre.Track(track_id, data_home, dataset_name, index, metadata)`

AcousticBrainz Genre Dataset track class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored. If *None*, looks for the data in the default directory, *~/mir_datasets*

Variables

- **track_id** (*str*) – track id
- **genre** (*list*) – human-labeled genre and subgenres list
- **mbid** (*str*) – musicbrainz id
- **mbid_group** (*str*) – musicbrainz id group
- **artist** (*list*) – the track's artist/s
- **title** (*list*) – the track's title
- **date** (*list*) – the track's release date/s
- **filename** (*str*) – the track's filename
- **album** (*list*) – the track's album/s
- **track_number** (*list*) – the track number/s

- **tonal** (*dict*) – dictionary of acousticbrainz tonal features
- **low_level** (*dict*) – dictionary of acousticbrainz low-level features
- **rhythm** (*dict*) – dictionary of acousticbrainz rhythm features

Other Parameters
acousticbrainz_metadata (*dict*) – dictionary of metadata provided by AcousticBrainz

property album

metadata album annotation

Returns
list – album

property artist

metadata artist annotation

Returns
list – artist

property date

metadata date annotation

Returns
list – date

property file_name

metadata file_name annotation

Returns
str – file name

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

property low_level

low_level track descriptors.

Returns

dict –

- 'average_loudness': dynamic range descriptor. It rescales average loudness, computed on 2sec windows with 1 sec overlap, into the [0,1] interval. The value of 0 corresponds to signals with large dynamic range, 1 corresponds to signal with little dynamic range. Algorithms: Loudness
- 'dynamic_complexity': dynamic complexity computed on 2sec windows with 1sec overlap. Algorithms: DynamicComplexity
- 'silence_rate_20dB', 'silence_rate_30dB', 'silence_rate_60dB': rate of silent frames in a signal for thresholds of 20, 30, and 60 dBs. Algorithms: SilenceRate
- 'spectral_rms': spectral RMS. Algorithms: RMS
- 'spectral_flux': spectral flux of a signal computed using L2-norm. Algorithms: Flux

- 'spectral_centroid', 'spectral_kurtosis', 'spectral_spread', 'spectral_skewness': centroid and central moments statistics describing the spectral shape. Algorithms: Centroid, CentralMoments
- 'spectral_rolloff': the roll-off frequency of a spectrum. Algorithms: RollOff
- 'spectral_decrease': spectral decrease. Algorithms: Decrease
- 'hfc': high frequency content descriptor as proposed by Masri. Algorithms: HFC
- 'zerocrossingrate' zero-crossing rate. Algorithms: ZeroCrossingRate
- 'spectral_energy': spectral energy. Algorithms: Energy
- 'spectral_energyband_low', 'spectral_energyband_middle_low', 'spectral_energyband_middle_high',
- 'spectral_energyband_high': spectral energy in frequency bands [20Hz, 150Hz], [150Hz, 800Hz], [800Hz, 4kHz], and [4kHz, 20kHz]. Algorithms EnergyBand
- 'barkbands': spectral energy in 27 Bark bands. Algorithms: BarkBands
- 'melbands': spectral energy in 40 mel bands. Algorithms: MFCC
- 'erbbands': spectral energy in 40 ERB bands. Algorithms: ERBBands
- 'mfcc': the first 13 mel frequency cepstrum coefficients. See algorithm: MFCC
- 'gfcc': the first 13 gammatone feature cepstrum coefficients. Algorithms: GFCC
- 'barkbands_crest', 'barkbands_flatness_db': crest and flatness computed over energies in Bark bands. Algorithms: Crest, FlatnessDB
- 'barkbands_kurtosis', 'barkbands_skewness', 'barkbands_spread': central moments statistics over energies in Bark bands. Algorithms: CentralMoments
- 'melbands_crest', 'melbands_flatness_db': crest and flatness computed over energies in mel bands. Algorithms: Crest, FlatnessDB
- 'melbands_kurtosis', 'melbands_skewness', 'melbands_spread': central moments statistics over energies in mel bands. Algorithms: CentralMoments
- 'erbbands_crest', 'erbbands_flatness_db': crest and flatness computed over energies in ERB bands. Algorithms: Crest, FlatnessDB
- 'erbbands_kurtosis', 'erbbands_skewness', 'erbbands_spread': central moments statistics over energies in ERB bands. Algorithms: CentralMoments
- 'dissonance': sensory dissonance of a spectrum. Algorithms: Dissonance
- 'spectral_entropy': Shannon entropy of a spectrum. Algorithms: Entropy
- 'pitch_salience': pitch salience of a spectrum. Algorithms: PitchSalience
- 'spectral_complexity': spectral complexity. Algorithms: SpectralComplexity
- 'spectral_contrast_coeffs', 'spectral_contrast_valleys': spectral contrast features. Algorithms: SpectralContrast

property rhythm

rhythm essentia extractor descriptors

Returns

dict –

- 'beats_position': time positions [sec] of detected beats using beat tracking algorithm by Degara et al., 2012. Algorithms: RhythmExtractor2013, BeatTrackerDegara
- 'beats_count': number of detected beats
- 'bpm': BPM value according to detected beats
- 'bpm_histogram_first_peak_bpm', 'bpm_histogram_first_peak_spread', 'bpm_histogram_first_peak_weight',
- 'bpm_histogram_second_peak_bpm', 'bpm_histogram_second_peak_spread', 'bpm_histogram_second_peak_weight': descriptors characterizing highest and second highest peak of the BPM histogram. Algorithms: BpmHistogramDescriptors
- 'beats_loudness', 'beats_loudness_band_ratio': spectral energy computed on beats segments of audio across the whole spectrum, and ratios of energy in 6 frequency bands. Algorithms: BeatsLoudness, SingleBeatLoudness
- 'onset_rate': number of detected onsets per second. Algorithms: OnsetRate
- 'danceability': danceability estimate. Algorithms: Danceability

property title

metadata title annotation

Returns

list – title

to_jams()

the track's data in jams format

Returns

jams.JAMS – return track data in jam format

property tonal

tonal features

Returns

dict –

- 'tuning_frequency': estimated tuning frequency [Hz]. Algorithms: TuningFrequency
- 'tuning_nontempered_energy_ratio' and 'tuning_equal_tempered_deviation'
- 'hpcp', 'thpcp': 32-dimensional harmonic pitch class profile (HPCP) and its transposed version. Algorithms: HPCP
- 'hpcp_entropy': Shannon entropy of a HPCP vector. Algorithms: Entropy
- 'key_key', 'key_scale': Global key feature. Algorithms: Key
- 'chords_key', 'chords_scale': Global key extracted from chords detection.
- 'chords_strength', 'chords_histogram': : strength of estimated chords and normalized histogram of their progression; Algorithms: ChordsDetection, ChordsDescriptors
- 'chords_changes_rate', 'chords_number_rate': chords change rate in the progression; ratio of different chords from the total number of chords in the progression; Algorithms: ChordsDetection, ChordsDescriptors

property tracknumber

metadata tracknumber annotation

Returns
list – tracknumber

mirdata.datasets.acousticbrainz_genre.**load_extractor**(*fhandle*)

Load a AcousticBrainz Dataset json file with all the features and metadata.

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to a json file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

2.5.2 baf

BAF Loader

Dataset Info

BAF dataset is only available upon request. To download the audio request access in this link: https://doi.org/10.5281/zenodo.6868083. Then unzip the audio into the baf general dataset folder for the rest of annotations and files. Please include, in the justification field, your academic affiliation (if you have one) and a brief description of your research topics and why you would like to use this dataset.

Overview

Broadcast Audio Fingerprinting dataset is an open, available upon request, annotated dataset for the task of music monitoring in broadcast. It contains 2,000 tracks from Epidemic Sound's private catalogue as reference tracks that represent 74 hours. As queries, it contains over 57 hours of TV broadcast audio from 23 countries and 203 channels distributed with 3,425 one-min audio excerpts.

It has been annotated by six annotators in total and each query has been cross-annotated by three of them obtaining high inter-annotator agreement percentages, which validates the annotation methodology and ensures the reliability of the annotations.

Purpose of the dataset

This dataset aims to become the standard dataset to evaluate Audio Fingerprinting algorithms since it's built on real data, without the use of any data-augmentation techniques. It is also the first dataset to address background music fingerprinting, which is a real problem in royalties distribution.

Dataset use

This dataset is available for conducting non-commercial research related to audio analysis. It shall not be used for music generation or music synthesis.

About the data

- Sampling frequency: 8 kHz
- Bit-depth: 16 bit
- Number of channels: 1
- Encoding: pcm_s16le
- Audio format: .wav

Annotations mark which tracks sound (either in foreground or background) in each query (if any) and also the specific times where it starts and ends sound in the query. Note that there are 88 queries that doesn't have any matches/annotations .

For more information check the dedicated Github repository: https://github.com/guillemcortes/baf-dataset and the dataset datasheet included in the files.

Ownership of the data

Next, we specify the ownership of all the data included in BAF: Broadcast Audio Fingerprinting dataset. For licensing information, please refer to the "License" section.

Reference tracks

The reference tracks are owned by Epidemic Sound AB, which has given a worldwide, revocable, non-exclusive, royalty-free licence to use and reproduce this data collection consisting of 2,000 low-quality monophonic 8kHz downsampled audio recordings.

Query tracks

The query tracks come from publicly available TV broadcast emissions so the ownership of each recording belongs to the channel that emitted the content. We publish them under the right of quotation provided by the Berne Convention.

Annotations

Guillem Cortès together with Alex Ciurana and Emilio Molina from BMAT Music Licensing S.L. have managed the annotation therefore the annotations belong to BMAT.

Accessing the dataset

The dataset is available upon request. Please include, in the justification field, your academic affiliation (if you have one) and a brief description of your research topics and why you would like to use this dataset. Bear in mind that this information is important for the evaluation of every access request.

License

```
Given the different ownership of the elements of the dataset, the
dataset is licensed under the following conditions:
    * User's access request
    * Research only, non-commercial purposes
    * No adaptations nor derivative works
    * Attribution to Epidemic Sound and the authors as it is indicated
        in the "citation" section.
```

Acknowledgments

With the support of Ministerio de Ciencia Innovación y universidades through Retos-Colaboración call, reference: RTC2019-007248-7, and also with the support of the Industrial Doctorates Plan of the Secretariat of Universities and Research of the Department of Business and Knowledge of the Generalitat de Catalunya. Reference: DI46-2020.

---

class mirdata.datasets.baf.**Dataset**(*data_home=None*, *version='default'*)

The BAF dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format

- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits`(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_multitracks`()

Load all multitracks in the dataset

Returns

dict – {*mtrack_id*: multitrack data}

Raises

`NotImplementedError` – If the dataset does not support Multitracks

`load_tracks`()

Load all tracks in the dataset

Returns

dict – {*track_id*: track data}

Raises

`NotImplementedError` – If the dataset does not support Tracks

mtrack_ids

Return track ids

Returns
list – A list of track ids

track_ids

Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.baf.**EventDataExtended**(*intervals*, *interval_unit*, *events*, *event_unit*, *tags*, *tag_unit*)

EventDataExtended class. Inherits from annotations.EventData class. An event is defined here as a match query-reference, and the time interval in the query. This class adds the possibility to attach tags to each event, useful if there's a need to differenciate them. In BAF, tags are [single, majority, unanimity].

Variables

- **tags** (*list*) – list of tag labels (as strings)
- **tag_unit** (*str*) – tag units, one of TAG_UNITS
- **intervals** (*np.ndarray*) – (n x 2) array of intervals
- **positive** (*in the form [start_time, end_time]. Times should be*) –
- **duration** (*and intervals should have non-negative*) –
- **interval_unit** (*str*) – unit of the time values in intervals. One
- **TIME_UNITS.** (*of*) –
- **interval_unit** – interval units, one of TIME_UNITS
- **events** (*list*) – list of event labels (as strings)
- **event_unit** (*str*) – event units, one of EVENT_UNITS

mirdata.datasets.baf.**TAG_UNITS = {'open': 'no scrict schema or units'}**

Tag units

class mirdata.datasets.baf.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

BAF track class.

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored. If *None*, looks for the data in the default directory, *~/mir_datasets/baf*

Variables
audio_path (*str*) – audio path

Properties:

audio (Tuple[np.ndarray, float]): audio array country (str): country of emission channel (str): tv channel of the emission datetime (str): datetime of the TV emission in YYYY-MM-DD HH:mm:ssformat matches (list): list of matches for a specific query

Returns

Track – BAF dataset track

property audio: Tuple[numpy.ndarray, float]

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters

key (*string*) – Index key of the audio or annotation type

Returns

str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns

jams.JAMS – the track's data in jams format

mirdata.datasets.baf.load_audio(*fpath: str*) → Tuple[numpy.ndarray, float]

Load a baf audio file.

Parameters

fpath (*str*) – path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.baf.load_matches(*track_metadata: dict*) → *EventDataExtended* | None

Load the matches corresponding to a query track.

Parameters

track_metadata (*dict*) – track's metadata

Returns

Optional[EventDataExtended] – Track's annotations in EvendDataExtended format

2.5.3 ballroom

Ballroom Rhythm Dataset Loader

Dataset Info

The Ballroom Rhythm Dataset is a comprehensive collection of rhythm annotations for ballroom dance music. This dataset is designed for tasks such as beat tracking, rhythm analysis, and tempo estimation in ballroom dance music. It includes annotations for beats and bars corresponding to different dance styles within the ballroom genre.

Dataset Overview:

The dataset offers beat and bar annotations for various ballroom dance styles, such as Waltz, Tango, Viennese Waltz, Slow Foxtrot, Quickstep, Samba, Cha-Cha-Cha, Rumba, Paso Doble, and Jive. These annotations are provided in a format that includes beat time in seconds and beat ID, facilitating precise rhythm analysis.

Beat and Bar Annotations:

The beat annotations are structured as *.beats* files, where each line represents a beat with its timestamp and beat ID. For example, a line *9.430022675 3* indicates that the third beat of a bar is located at 9.43 seconds. This format is particularly useful for identifying downbeats, as they correspond to beats with ID = 1.

Annotation Methodology:

The dataset's annotations are based on the tempo guidelines of each ballroom dance style. Initial annotations were generated using a beat tracker, and then manually adjusted for accuracy. This method ensures that the annotations reflect the characteristic rhythms of each dance style.

Applications:

The Ballroom Rhythm Dataset is ideal for developing and testing algorithms for beat tracking, tempo estimation, and rhythm analysis in ballroom dance music. It can also be used for educational purposes, offering insights into the rhythmic structures of various ballroom dance styles.

Acknowledgments and References:

This dataset was created with the collaboration of experts in ballroom dance music. We extend our gratitude to those who contributed their knowledge and expertise to this project. For detailed information on the dataset and its creation, please refer to the associated research papers and documentation.

[1] Gouyon F., A. Klapuri, S. Dixon, M. Alonso, G. Tzanetakis, C. Uhle, and P. Cano. An experimental comparison of audio tempo induction algorithms. Transactions on Audio, Speech and Language Processing 14(5), pp.1832-1844, 2006.

[2] Böck, S., and M. Schedl. Enhanced beat tracking with context-aware neural networks. In Proceedings of the International Conference on Digital Audio Effects (DAFX), 2010.

[3] Dixon, S., F. Gouyon & G. Widmer. Towards Characterisation of Music via Rhythmic Patterns. In Proceedings of the 5th International Society for Music Information Retrieval Conference (ISMIR). 2004.

class mirdata.datasets.ballroom.**Dataset**(*data_home=None*, *version='default'*)

The ballroom dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format

- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits`(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_multitracks`()

Load all multitracks in the dataset

Returns

dict – {*mtrack_id*: multitrack data}

Raises

`NotImplementedError` – If the dataset does not support Multitracks

`load_tracks`()

Load all tracks in the dataset

Returns

dict – {*track_id*: track data}

Raises

`NotImplementedError` – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns
list – A list of track ids

`track_ids`

Return track ids

Returns
list – A list of track ids

`validate`(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

`class mirdata.datasets.ballroom.Track`(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

Ballroom Rhythm class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored. default=None If *None*, looks for the data in the default directory, *~/mir_datasets*

Variables

- **audio_path** (*str*) – path to audio file
- **beats_path** (*str*) – path to beats file
- **tempo_path** (*str*) – path to tempo file

Other Parameters

- **beats** (*BeatData*) – human-labeled beat annotations
- **tempo** (*float*) – human-labeled tempo annotations

`property audio: Tuple[numpy.ndarray, float] | None`

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

`get_path`(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

`to_jams()`

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

`mirdata.datasets.ballroom.load_audio(fhandle: BinaryIO) → Tuple[numpy.ndarray, float]`

Load a Ballroom audio file.

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to an audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

`mirdata.datasets.ballroom.load_beats(fhandle: TextIO)`

Load beats

Parameters
fhandle (*str or file-like*) – Local path where the beats annotation is stored.

Returns
BeatData – beat annotations

`mirdata.datasets.ballroom.load_tempo(fhandle: TextIO) → float`

Load tempo

Parameters
fhandle (*str or file-like*) – Local path where the tempo annotation is stored.

Returns
float – tempo annotation

2.5.4 beatles

Beatles Dataset Loader

Dataset Info

The Beatles Dataset includes beat and metric position, chord, key, and segmentation annotations for 179 Beatles songs. Details can be found in https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.207.4076&rep=rep1&type=pdf and http://isophonics.net/content/reference-annotations-beatles.

`class mirdata.datasets.beatles.Dataset(data_home=None, version='default')`

The beatles dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –

- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits`(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_audio`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.beatles.load_audio

`load_beats`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.beatles.load_beats

`load_chords`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.beatles.load_chords

`load_multitracks`()

Load all multitracks in the dataset

Returns

dict – {*mtrack_id*: multitrack data}

Raises

`NotImplementedError` – If the dataset does not support Multitracks

load_sections(*args, **kwargs)

Deprecated since version 0.3.4: Use mirdata.datasets.beatles.load_sections

load_tracks()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

mtrack_ids

Return track ids

Returns
list – A list of track ids

track_ids

Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.beatles.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

Beatles track class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – path where the data lives

Variables

- **audio_path** (*str*) – track audio path
- **beats_path** (*str*) – beat annotation path
- **chords_path** (*str*) – chord annotation path
- **keys_path** (*str*) – key annotation path
- **sections_path** (*str*) – sections annotation path
- **title** (*str*) – title of the track
- **track_id** (*str*) – track id

Other Parameters

- **beats** (*BeatData*) – human-labeled beat annotations
- **chords** (*ChordData*) – human-labeled chord annotations

- **key** (*KeyData*) – local key annotations
- **sections** (*SectionData*) – section annotations

property audio: Tuple[numpy.ndarray, float] | None

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

the track's data in jams format

Returns
jams.JAMS – return track data in jam format

mirdata.datasets.beatles.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a Beatles audio file.

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to an audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.beatles.**load_beats**(*fhandle: TextIO*) → *BeatData*

Load Beatles format beat data from a file

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to a beat annotation file

Returns
BeatData – loaded beat data

mirdata.datasets.beatles.**load_chords**(*fhandle: TextIO*) → *ChordData*

Load Beatles format chord data from a file

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to a chord annotation file

Returns
ChordData – loaded chord data

mirdata.datasets.beatles.**load_key**(*fhandle: TextIO*) → *KeyData*

Load Beatles format key data from a file

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to a key annotation file

Returns
KeyData – loaded key data

mirdata.datasets.beatles.**load_sections**(*fhandle: TextIO*) → *SectionData*

Load Beatles format section data from a file

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to a section annotation file

Returns
SectionData – loaded section data

2.5.5 beatport_key

beatport_key Dataset Loader

Dataset Info

The Beatport EDM Key Dataset includes 1486 two-minute sound excerpts from various EDM subgenres, annotated with single-key labels, comments and confidence levels generously provided by Eduard Mas Marín, and thoroughly revised and expanded by Ángel Faraldo.

The original audio samples belong to online audio snippets from Beatport, an online music store for DJ's and Electronic Dance Music Producers (<http:www.beatport.com>). If this dataset were used in further research, we would appreciate the citation of the current DOI (10.5281/zenodo.1101082) and the following doctoral dissertation, where a detailed description of the properties of this dataset can be found:

```
Ángel Faraldo (2017). Tonality Estimation in Electronic Dance Music: A Computational and␣
↪Musically Informed
Examination. PhD Thesis. Universitat Pompeu Fabra, Barcelona.
```

This dataset is mainly intended to assess the performance of computational key estimation algorithms in electronic dance music subgenres.

Data License: Creative Commons Attribution Share Alike 4.0 International

---

class mirdata.datasets.beatport_key.**Dataset**(*data_home=None*, *version='default'*)

The beatport_key dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

`choice_multitrack()`

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

`choice_track()`

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

`cite()`

Print the reference

`property default_path`

Get the default path for the dataset

Returns
str – Local path to the dataset

`download(partial_download=None, force_overwrite=False, cleanup=False)`

Download the dataset

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- `ValueError` – if invalid keys are passed to partial_download
- `IOError` – if a downloaded file's checksum is different from expected

`get_mtrack_splits()`

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- `AttributeError` – If this dataset does not have multitracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits(splits, seed=42, split_names=None)`

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_artist`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.beatport_key.load_artist

`load_audio`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.beatport_key.load_audio

`load_genre`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.beatport_key.load_genre

`load_key`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.beatport_key.load_key

`load_multitracks`()

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

`load_tempo`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.beatport_key.load_tempo

`load_tracks`()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

mtrack_ids

Return track ids

Returns
list – A list of track ids

track_ids

Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.beatport_key.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

beatport_key track class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored.

Variables

- **audio_path** (*str*) – track audio path
- **keys_path** (*str*) – key annotation path
- **metadata_path** (*str*) – sections annotation path
- **title** (*str*) – title of the track
- **track_id** (*str*) – track id

Other Parameters

- **key** (*list*) – list of annotated musical keys
- **artists** (*list*) – artists involved in the track
- **genre** (*dict*) – genres and subgenres
- **tempo** (*int*) – tempo in beats per minute

property audio

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.beatport_key.load_artist(*fhandle*)

Load beatport_key tempo data from a file

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to metadata file

Returns
list – list of artists involved in the track.

mirdata.datasets.beatport_key.load_audio(*fpath*)

Load a beatport_key audio file.

Parameters
fpath (*str*) – path to an audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.beatport_key.load_genre(*fhandle*)

Load beatport_key genre data from a file

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to metadata file

Returns
dict – with the list with genres ['genres'] and list with sub-genres ['sub_genres']

mirdata.datasets.beatport_key.load_key(*fhandle*)

Load beatport_key format key data from a file

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to a key annotation file

Returns
list – list of annotated keys

mirdata.datasets.beatport_key.load_tempo(*fhandle*)

Load beatport_key tempo data from a file

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to metadata file

Returns
str – tempo in beats per minute

2.5.6 billboard

McGill Billboard Dataset Loader

Dataset Info

The McGill Billboard dataset includes annotations and audio features corresponding to 890 slots from a random sample of Billboard chart slots. It also includes metadata like Billboard chart date, peak rank, artist name, etc. Details can be found at https://ddmal.music.mcgill.ca/research/The_McGill_Billboard_Project_(Chord_Analysis_Dataset)

class mirdata.datasets.billboard.**Dataset**(*data_home=None*, *version='default'*)

The McGill Billboard dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns

str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded

- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

get_random_track_splits(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

get_track_splits()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- **AttributeError** – If this dataset does not have tracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license()`

Print the license

`load_audio(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.billboard.load_audio

`load_chords(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.billboard.load_chords

`load_multitracks()`

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

`load_named_sections(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.billboard.load_named_sections

`load_sections(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.billboard.load_sections

`load_tracks()`

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
`NotImplementedError` – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns
list – A list of track ids

`track_ids`

Return track ids

Returns
list – A list of track ids

`validate(verbose=True)`

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.billboard.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

McGill Billboard Dataset Track class

Parameters

track_id (*str*) – track id of the track

Variables

- **track_id** (*str*) – the index for the sample entry
- **audio_path** (*str*) – audio path of the track
- **date** (*chart*) – the date of the chart for the entry
- **rank** (*peak*) – the desired rank on that chart
- **rank** – the rank of the song actually annotated, which may be up to 2 ranks higher or lower than the target rank
- **title** (*str*) – the title of the song annotated
- **artist** (*str*) – the name of the artist performing the song annotated
- **rank** – the highest rank the song annotated ever achieved on the Billboard Hot 100
- **chart** (*weeks on*) – the number of weeks the song annotated spent on the Billboard Hot 100 chart in total

Other Parameters

- **chords_full** (*ChordData*) – HTK-style LAB files for the chord annotations (full)
- **chords_majmin7** (*ChordData*) – HTK-style LAB files for the chord annotations (majmin7)
- **chords_majmin7inv** (*ChordData*) – HTK-style LAB files for the chord annotations (majmin7inv)
- **chords_majmin** (*ChordData*) – HTK-style LAB files for the chord annotations (majmin)
- **chords_majmininv** (*ChordData*) – HTK-style LAB files for the chord annotations(majmininv)
- **chroma** (*np.array*) – Array containing the non-negative-least-squares chroma vectors
- **tuning** (*list*) – List containing the tuning estimates
- **sections** (*SectionData*) – Letter-annotated section data (A,B,A')
- **named_sections** (*SectionData*) – Name-annotated section data (intro, verse, chorus)
- **salami_metadata** (*dict*) – Metadata of the Salami LAB file

property **audio**: Tuple[numpy.ndarray, float] | None

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

chroma

Non-negative-least-squares (NNLS) chroma vectors from the Chordino Vamp plug-in

Returns

np.ndarray - NNLS chroma vector

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

tuning

Tuning estimates from the Chordino Vamp plug-in

Returns
list - list of of tuning estimates []

mirdata.datasets.billboard.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a Billboard audio file.

Parameters
fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.billboard.**load_chords**(*fhandle: TextIO*)

Load chords from a Salami LAB file.

Parameters
fhandle (*str or file-like*) – path to audio file

Returns
ChordData – chord data

mirdata.datasets.billboard.**load_named_sections**(*fpath: str*)

Load name-annotated sections from a Salami LAB file.

Parameters
fpath (*str*) – path to sections file

Returns
SectionData – section data

mirdata.datasets.billboard.**load_sections**(*fpath: str*)

Load letter-annotated sections from a Salami LAB file.

Parameters
fpath (*str*) – path to sections file

Returns
SectionData – section data

2.5.7 candombe

Candombe Dataset Loader

Dataset Info

This is a dataset of Candombe recordings with annotated beats and downbeats, totaling over 2 hours of audio. It comprises 35 complete performances by renowned players, in groups of three to five drums. Recording sessions were conducted in studio, in the context of musicological research over the past two decades. A total of 26 tambor players took part, belonging to different generations and representing all the important traditional Candombe styles. The audio files are stereo with a sampling rate of 44.1 kHz and 16-bit precision. The location of beats and downbeats was annotated by an expert, adding to more than 4700 downbeats.

The audio is provided as .flac files and the annotations as .csv files. The values in the first column of the csv file are the time instants of the beats. The numbers on the second column indicate both the bar number and the beat number within the bar. For instance, 1.1, 1.2, 1.3 and 1.4 are the four beats of the first bar. Hence, each label ending with .1 indicates a downbeat. Another set of annotations are provided as .beats files in which the bar numbers are removed.

`class mirdata.datasets.candombe.Dataset(data_home=None, version='default')`

The candombe dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

`choice_multitrack()`

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

`choice_track()`

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

`cite()`

Print the reference

`property default_path`

Get the default path for the dataset

Returns

str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

get_random_track_splits(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_track_splits()`

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of track_ids

`license()`

Print the license

`load_multitracks()`

Load all multitracks in the dataset

Returns

dict – {*mtrack_id*: multitrack data}

Raises

`NotImplementedError` – If the dataset does not support Multitracks

`load_tracks()`

Load all tracks in the dataset

Returns

dict – {*track_id*: track data}

Raises

`NotImplementedError` – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns

list – A list of track ids

`track_ids`

Return track ids

Returns

list – A list of track ids

`validate(verbose=True)`

Validate if the stored dataset is a valid version

Parameters

verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

`class mirdata.datasets.candombe.Track(track_id, data_home, dataset_name, index, metadata)`

Candombe Track class

Parameters

track_id (*str*) – track id of the track

Variables

- **audio_path** (*str*) – path to audio file
- **beats_path** (*str*) – path to beats file

Other Parameters
beats (*BeatData*) – beat annotations

property audio: Tuple[numpy.ndarray, float] | None
The track's audio

Returns
- np.ndarray - audio signal
- float - sample rate

beats
The track's beats

Returns
BeatData – loaded beat data

get_path(*key*)
Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()
Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.candombe.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]
Load a candombe audio file.

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to an audio file

Returns
- np.ndarray - the audio signal
- float - The sample rate of the audio file

mirdata.datasets.candombe.**load_beats**(*fhandle: TextIO*) → *BeatData*
Load a candombe beats file.

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to an audio file

Returns
BeatData – loaded beat data

2.5.8 cante100

cante100 Loader

Dataset Info

The cante100 dataset contains 100 tracks taken from the COFLA corpus. We defined 10 style families of which 10 tracks each are included. Apart from the style family, we manually annotated the sections of the track in which the vocals are present. In addition, we provide a number of low-level descriptors and the fundamental frequency corresponding to the predominant melody for each track. The meta-information includes editoral meta-data and the musicBrainz ID.

Total tracks: 100

cante100 audio is only available upon request. To download the audio request access in this link: https://zenodo.org/record/1324183. Then unzip the audio into the cante100 general dataset folder for the rest of annotations and files.

Audio specifications:

- Sampling frequency: 44.1 kHz
- Bit-depth: 16 bit
- Audio format: .mp3

cante100 dataset has spectrogram available, in csv format. spectrogram is available to download without request needed, so at first instance, cante100 loader uses the spectrogram of the tracks.

The available annotations are:

- F0 (predominant melody)
- Automatic transcription of notes (of singing voice)

CANTE100 LICENSE (COPIED FROM ZENODO PAGE)

```
The provided datasets are offered free of charge for internal non-commercial use.
We do not grant any rights for redistribution or modification. All data collections were␣
↪gathered
by the COFLA team.
© COFLA 2015. All rights reserved.
```

For more details, please visit: http://www.cofla-project.com/?page_id=134

class mirdata.datasets.cante100.**Dataset**(*data_home=None*, *version='default'*)

The cante100 dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns

str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list

- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_audio`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.cante100.load_audio

`load_melody`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.cante100.load_melody

`load_multitracks`()

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

`load_notes`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.cante100.load_notes

`load_spectrogram`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.cante100.load_spectrogram

load_tracks()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

mtrack_ids

Return track ids

Returns
list – A list of track ids

track_ids

Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.cante100.Track(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

cante100 track class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored. If *None*, looks for the data in the default directory, *~/mir_datasets/cante100*

Variables

- **track_id** (*str*) – track id
- **identifier** (*str*) – musicbrainz id of the track
- **artist** (*str*) – performing artists
- **title** (*str*) – title of the track song
- **release** (*str*) – release where the track can be found
- **duration** (*str*) – duration in seconds of the track

Other Parameters

- **melody** (*F0Data*) – annotated melody
- **notes** (*NoteData*) – annotated notes

property audio: Tuple[numpy.ndarray, float]

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

property spectrogram: numpy.ndarray | None

spectrogram of The track's audio

Returns
np.ndarray – spectrogram

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.cante100.load_audio(*fpath: str*) → Tuple[numpy.ndarray, float]

Load a cante100 audio file.

Parameters
fpath (*str*) – path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.cante100.load_melody(*fhandle: TextIO*) → *F0Data* | None

Load cante100 f0 annotations

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to melody annotation file

Returns
F0Data – predominant melody

mirdata.datasets.cante100.load_notes(*fhandle: TextIO*) → *NoteData*

Load note data from the annotation files

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to a notes annotation file

Returns
NoteData – note annotations

mirdata.datasets.cante100.load_spectrogram(*fhandle: TextIO*) → numpy.ndarray

Load a cante100 dataset spectrogram file.

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to an audio file

Returns
np.ndarray – spectrogram

2.5.9 cipi

Can I play it? (CIPI) Dataset Loader

Dataset Info

The "Can I Play It?" (CIPI) dataset is a specialized collection of 652 classical piano scores, provided in a machine-readable MusicXML format and accompanied by integer-based difficulty levels ranging from 1 to 9, as verified by expert pianists. Then, it provides embeddings for fingering and expresiveness of the piece. Each recording has multiple scores corresponding to it. This dataset focuses exclusively on classical piano music, offering a rich resource for music researchers, educators, and students. Developed by the Music Technology Group in Barcelona, by P. Ramoneda et al.

The CIPI dataset facilitates various applications such as the study of musical complexity, the selection of appropriately leveled pieces for students, and general research in music education. The dataset, alongside embeddings of multiple dimensions of difficulty, has been made publicly available to encourage ongoing innovation and collaboration within the music education and research communities.

The dataset has been published alongside a paper in Expert Systems with Applications Journal.

The dataset is shared under a Creative Commons Attribution Non Commercial Share Alike 4.0 International License, but need to be requested. Please do request the dataset here: https://zenodo.org/records/8037327. The dataset can only be used for open research purposes.

`class mirdata.datasets.cipi.Dataset(data_home=None, version='default')`

The Can I play it? (CIPI) dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

`choice_multitrack()`

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_multitracks`()

Load all multitracks in the dataset

Returns

dict – {*mtrack_id*: multitrack data}

Raises

`NotImplementedError` – If the dataset does not support Multitracks

`load_tracks`()

Load all tracks in the dataset

Returns

dict – {*track_id*: track data}

Raises

`NotImplementedError` – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns

list – A list of track ids

`track_ids`

Return track ids

Returns

list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.cipi.Track(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

Can I play it? (CIPI) track class

Parameters
track_id (*str*) – track id of the track

Variables

- **title** (*str*) – title of the track
- **book** (*str*) – book of the track
- **URI** (*str*) – URI of the track
- **composer** (*str*) – name of the author of the track
- **track_id** (*str*) – track id
- **musicxml_paths** (*list*) – path to musicxml score. If the music piece contains multiple movents the list will contain multiple paths.
- **difficulty_annotation** (*int*) – annotated difficulty
- **fingering_path** (*tuple*) – Path of fingering features from technique dimension computed with ArGNN fingering model. Return of two paths, the right hand and the ones of the left hand. Use torch.load(...) for loading the embeddings.
- **expressiveness_path** (*str*) – Path of expressiveness features from sound dimension computed with virtuosoNet model.Use torch.load(...) for loading the embeddings.
- **notes_path** (*str*) – Path of note features from notation dimension. Use torch.load(...) for loading the embeddings.

Other Parameters
scores (*list[music21.stream.Score]*) – music21 scores. If the work is splited in several movements the list will contain multiple scores.

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.cipi.**load_score**(*fhandle: str*, *data_home: str = 'tests/resources/mir_datasets/cipi'*) → music21.stream.Score

Load cipi score in music21 stream

Parameters

- **fhandle** (*str*) – path to MusicXML score
- **data_home** (*str*) – path to cipi dataset

Returns

music21.stream.Score – score in music21 format

2.5.10 compmusic_carnatic_rhythm

CompMusic Carnatic Rhythm Dataset Loader

Dataset Info

CompMusic Carnatic Rhythm Dataset is a rhythm annotated test corpus for automatic rhythm analysis tasks in Carnatic Music. The collection consists of audio excerpts from the CompMusic Carnatic research corpus, manually annotated time aligned markers indicating the progression through the taala cycle, and the associated taala related metadata. A brief description of the dataset is provided below. For a brief overview and audio examples of taalas in Carnatic music, please see: http://compmusic.upf.edu/examples-taala-carnatic

The dataset contains the following data:

AUDIO: The pieces are chosen from the CompMusic Carnatic music collection. The pieces were chosen in four popular taalas of Carnatic music, which encompasses a majority of Carnatic music. The pieces were chosen include a mix of vocal and instrumental recordings, new and old recordings, and to span a wide variety of forms. All pieces have a percussion accompaniment, predominantly Mridangam. The excerpts are full length pieces or a part of the full length pieces. There are also several different pieces by the same artist (or release group), and multiple instances of the same composition rendered by different artists. Each piece is uniquely identified using the MBID of the recording. The pieces are stereo, 160 kbps, mp3 files sampled at 44.1 kHz.

SAMA AND BEATS: The primary annotations are audio synchronized time-stamps indicating the different metrical positions in the taala cycle. The annotations were created using Sonic Visualizer by tapping to music and manually correcting the taps. Each annotation has a time-stamp and an associated numeric label that indicates the position of the beat marker in the taala cycle. The marked positions in the taala cycle are shown with numbers, along with the corresponding label used. In each case, the sama (the start of the cycle, analogous to the downbeat) are indicated using the numeral 1.

METADATA: For each excerpt, the taala of the piece, edupu (offset of the start of the piece, relative to the sama, measured in aksharas) of the composition, and the kalai (the cycle length scaling factor) are recorded. Each excerpt can be uniquely identified and located with the MBID of the recording, and the relative start and end times of the excerpt within the whole recording. A separate 5 digit taala based unique ID is also provided for each excerpt as a double check. The artist, release, the lead instrument, and the raaga of the piece are additional editorial metadata obtained from the release. A flag indicates if the excerpt is a full piece or only a part of a full piece. There are optional comments on audio quality and annotation specifics.

Possible uses of the dataset: Possible tasks where the dataset can be used include taala, sama and beat tracking, tempo estimation and tracking, taala recognition, rhythm based segmentation of musical audio, structural segmentation, audio to score/lyrics alignment, and rhythmic pattern discovery.

Dataset organization: The dataset consists of audio, annotations, an accompanying spreadsheet providing additional metadata. For a detailed description of the organization, please see the README in the dataset.

Data Subset: A subset of this dataset consisting of 118 two minute excerpts of music is also available. The content in the subset is equaivalent and is separately distributed for a quicker testing of algorithms and approaches.

The annotations files of this dataset are shared with the following license: Creative Commons Attribution Non Commercial Share Alike 4.0 International

---

class mirdata.datasets.compmusic_carnatic_rhythm.**Dataset**(*data_home=None*, *version='default'*)

The compmusic_carnatic_rhythm dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns

str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

`get_mtrack_splits()`

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits(splits, seed=42, split_names=None)`

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_random_track_splits(splits, seed=42, split_names=None)`

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_track_splits()`

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- **AttributeError** – If this dataset does not have tracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of track_ids

`license()`

Print the license

`load_multitracks()`

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
NotImplementedError – If the dataset does not support Multitracks

`load_tracks()`

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns
list – A list of track ids

`track_ids`

Return track ids

Returns
list – A list of track ids

`validate(verbose=True)`

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

`class mirdata.datasets.compmusic_carnatic_rhythm.Track(track_id, data_home, dataset_name, index, metadata)`

CompMusic Carnatic Music Rhythm class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored. default=None If *None*, looks for the data in the default directory, *~/mir_datasets*

Variables

- **audio_path** (*str*) – path to audio file
- **beats_path** (*srt*) – path to beats file
- **meter_path** (*srt*) – path to meter file

Other Parameters

- **beats** (*BeatData*) – beats annotation
- **meter** (*string*) – meter annotation
- **mbid** (*string*) – MusicBrainz ID
- **name** (*string*) – name of the recording in the dataset
- **artist** (*string*) – artists name
- **release** (*string*) – release name
- **lead_instrument_code** (*string*) – code for the load instrument
- **taala** (*string*) – taala annotation
- **raaga** (*string*) – raaga annotation
- **num_of_beats** (*int*) – number of beats in annotation
- **num_of_samas** (*int*) – number of samas in annotation

`property audio`

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

`get_path`(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

`to_jams`()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.compmusic_carnatic_rhythm.**load_audio**(*audio_path*)

Load an audio file.

Parameters
audio_path (*str*) – path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.compmusic_carnatic_rhythm.**load_beats**(*fhandle*)

Load beats

Parameters
fhandle (*str or file-like*) – Local path where the beats annotation is stored.

Returns
BeatData – beat annotations

mirdata.datasets.compmusic_carnatic_rhythm.**load_meter**(*fhandle*)

Load meter

Parameters
fhandle (*str or file-like*) – Local path where the meter annotation is stored.

Returns
float – meter annotation

2.5.11 compmusic_hindustani_rhythm

CompMusic Hindustani Rhythm Dataset Loader

Dataset Info

CompMusic Hindustani Rhythm Dataset is a rhythm annotated test corpus for automatic rhythm analysis tasks in Hindustani Music. The collection consists of audio excerpts from the CompMusic Hindustani research corpus, manually annotated time aligned markers indicating the progression through the taal cycle, and the associated taal related metadata. A brief description of the dataset is provided below.

For a brief overview and audio examples of taals in Hindustani music, please see: http://compmusic.upf.edu/examples-taal-hindustani

The dataset contains the following data:

AUDIO: The pieces are chosen from the CompMusic Hindustani music collection. The pieces were chosen in four popular taals of Hindustani music, which encompasses a majority of Hindustani khyal music. The pieces were chosen include a mix of vocal and instrumental recordings, new and old recordings, and to span three lays. For each taal, there are pieces in dhrut (fast), madhya (medium) and vilambit (slow) lays (tempo class). All pieces have Tabla as the percussion accompaniment. The excerpts are two minutes long. Each piece is uniquely identified using the MBID of the recording. The pieces are stereo, 160 kbps, mp3 files sampled at 44.1 kHz. The audio is also available as wav files for experiments.

SAM, VIBHAAG AND THE MAATRAS: The primary annotations are audio synchronized time-stamps indicating the different metrical positions in the taal cycle. The sam and matras of the cycle are annotated. The annotations were created using Sonic Visualizer by tapping to music and manually correcting the taps. Each annotation has a time-stamp and an associated numeric label that indicates the position of the beat marker in the taala cycle. The annotations and the associated metadata have been verified for correctness and completeness by a professional Hindustani musician and musicologist. The long thick lines show vibhaag boundaries. The numerals indicate the matra number in cycle. In each case, the sam (the start of the cycle, analogous to the downbeat) are indicated using the numeral 1.

METADATA: For each excerpt, the taal and the lay of the piece are recorded. Each excerpt can be uniquely identified and located with the MBID of the recording, and the relative start and end times of the excerpt within the whole recording. A separate 5 digit taal based unique ID is also provided for each excerpt as a double check. The artist, release, the lead instrument, and the raag of the piece are additional editorial metadata obtained from the release. There are optional comments on audio quality and annotation specifics.

The dataset consists of excerpts with a wide tempo range from 10 MPM (matras per minute) to 370 MPM. To study any effects of the tempo class, the full dataset (HMDf) is also divided into two other subsets - the long cycle subset (HMDl) consisting of vilambit (slow) pieces with a median tempo between 10-60 MPM, and the short cycle subset (HMDs) with madhyalay (medium, 60-150 MPM) and the drut lay (fast, 150+ MPM).

Possible uses of the dataset: Possible tasks where the dataset can be used include taal, sama and beat tracking, tempo estimation and tracking, taal recognition, rhythm based segmentation of musical audio, audio to score/lyrics alignment, and rhythmic pattern discovery.

Dataset organization: The dataset consists of audio, annotations, an accompanying spreadsheet providing additional metadata, a MAT-file that has identical information as the spreadsheet, and a dataset description document.

The annotations files of this dataset are shared with the following license: Creative Commons Attribution Non Commercial Share Alike 4.0 International

---

class mirdata.datasets.compmusic_hindustani_rhythm.**Dataset**(*data_home=None*, *version='default'*)

The compmusic_hindustani_rhythm dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

`get_mtrack_splits()`

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits(splits, seed=42, split_names=None)`

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_random_track_splits(splits, seed=42, split_names=None)`

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_track_splits()`

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- **AttributeError** – If this dataset does not have tracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of track_ids

`license()`

Print the license

`load_multitracks()`

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
NotImplementedError – If the dataset does not support Multitracks

`load_tracks()`

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns
list – A list of track ids

`track_ids`

Return track ids

Returns
list – A list of track ids

`validate(verbose=True)`

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

`class mirdata.datasets.compmusic_hindustani_rhythm.Track(track_id, data_home, dataset_name, index, metadata)`

CompMusic Hindustani Music Rhythm class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored. default=None If *None*, looks for the data in the default directory, *~/mir_datasets*

Variables

- **audio_path** (*str*) – path to audio file
- **beats_path** (*srt*) – path to beats file
- **meter_path** (*srt*) – path to meter file

Other Parameters

- **beats** (*BeatData*) – beats annotation
- **meter** (*string*) – meter annotation
- **mbid** (*string*) – MusicBrainz ID
- **name** (*string*) – name of the recording in the dataset
- **artist** (*string*) – artists name
- **release** (*string*) – release name
- **lead_instrument_code** (*string*) – code for the load instrument
- **taala** (*string*) – taala annotation
- **raaga** (*string*) – raaga annotation
- **laya** (*string*) – laya annotation
- **num_of_beats** (*int*) – number of beats in annotation
- **num_of_samas** (*int*) – number of samas in annotation
- **median_matra_period** (*float*) – median matra per period
- **median_matras_per_min** (*float*) – median matras per minute
- **median_ISI** (*float*) – median ISI
- **median_avarts_per_min** (*float*) – median avarts per minute

`property audio`

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

`get_path`(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

`to_jams`()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

`mirdata.datasets.compmusic_hindustani_rhythm.load_audio`(*audio_path*)

Load an audio file.

Parameters
audio_path (*str*) – path to audio file

Returns

- np.ndarray - the mono audio signal

- float - The sample rate of the audio file

mirdata.datasets.compmusic_hindustani_rhythm.**load_beats**(*fhandle*)

Load beats

Parameters
fhandle (*str or file-like*) – Local path where the beats annotation is stored.

Returns
BeatData – beat annotations

mirdata.datasets.compmusic_hindustani_rhythm.**load_meter**(*fhandle*)

Load meter

Parameters
fhandle (*str or file-like*) – Local path where the meter annotation is stored.

Returns
float – meter annotation

2.5.12 compmusic_indian_tonic

Indian Art Music Tonic Loader

Dataset Info

This loader includes a combination of six different datasets for the task of Indian Art Music tonic identification.

These datasets comprise audio excerpts and manually done annotations of the tonic pitch of the lead artist for each audio excerpt. Each excerpt is accompanied by its associated editorial metadata. These datasets can be used to develop and evaluate computational approaches for automatic tonic identification in Indian art music. These datasets have been used in several articles mentioned below. A majority of These datasets come from the CompMusic corpora of Indian art music, for which each recording is associated with a MBID. Through the MBID other information can be obtained using the Dunya API.

These six datasets are used for for the task of tonic identification for Indian Art Music, and can be used for a comparative evaluation. To the best of our knowledge these are the largest datasets available for tonic identification for Indian art music. These datases vary in terms of the audio quality, recording period (decade), the number of recordings for Carnatic, Hindustani, male and female singers and instrumental and vocal excerpts.

All the datasets (annotations) are version controlled. The audio files corresponding to these datsets are made available on request for only research purposes. See DOWNLOAD_INFO of this loader.

The tonic annotations are availabe both in tsv and json format. The loader uses the JSON formatted annotations.

```
'ID': {
    'artist': <name of the lead artist if available>,
    'filepath': <relative path to the audio file>,
    'gender': <gender of the lead singer if available>,
    'mbid': <musicbrainz id when available>,
    'tonic': <tonic in Hz>,
    'tradition': <Hindustani or Carnatic>,
    'type': <vocal or instrumental>
}
```

where keys of the main dictionary are the filepaths to the audio files (feature path is exactly the same with a different extension of the file name).

Despite not being loaded in this dataloader, the dataset includes features, which may be integrated to the loader in future releases. However these features may be easily computed following the instructions in the related paper. See BIBTEX.

There are a total of 2161 audio excerpts, and while the CM collection includes aproximately 50% Carnatic and 50% Hindustani recordings, IITM and IISc collections are 100% Carnatic music. The excerpts vary a lot in duration. See [this webpage](https://compmusic.upf.edu/iam-tonic-dataset) for a detailed overview of the datasets.

If you have any questions or comments about the dataset, please feel free to email: [sankalp (dot) gulati (at) gmail (dot) com], or [sankalp (dot) gulati (at) upf (dot) edu].

---

class mirdata.datasets.compmusic_indian_tonic.**Dataset**(*data_home=None*, *version='default'*)

The compmusic_indian_tonic dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns

str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.

- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

get_random_track_splits(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

get_track_splits()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- **AttributeError** – If this dataset does not have tracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license()`

Print the license

`load_multitracks()`

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

`load_tracks()`

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
`NotImplementedError` – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns
list – A list of track ids

`track_ids`

Return track ids

Returns
list – A list of track ids

`validate(verbose=True)`

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

`class mirdata.datasets.compmusic_indian_tonic.Track(track_id, data_home, dataset_name, index, metadata)`

CompMusic Tonic Dataset track class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored.

Variables

- `track_id` (*str*) – track id
- `audio_path` (*str*) – audio path

Other Parameters

- **tonic** (*float*) – tonic annotation
- **artist** (*str*) – performing artist
- **gender** (*str*) – gender of the recording artists
- **mbid** (*str*) – MusicBrainz ID of the piece (if available)
- **type** (*str*) – type of piece (vocal, instrumental, etc.)
- **tradition** (*str*) – tradition of the piece (Carnatic or Hindustani)

`property audio`

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

`get_path`(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

`to_jams`()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

`mirdata.datasets.compmusic_indian_tonic.load_audio`(*audio_path*)

Load a Indian Art Music Tonic audio file.

Parameters
fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

2.5.13 compmusic_jingju_acappella

Jingju A Cappella Singing Dataset Loader

Dataset Info

Description:
This dataset is a collection of boundary annotations of a cappella singing performed by Beijing Opera (Jingju,) professional and amateur singers.

Contents:

1. wav.zip: audio files in .wav format, mono or stereo.

2. pycode.zip: util code for parsing the .textgrid annotation
3. catalogue*.csv: recording metadata, source separation recordings are not included.
4. annotation_txt.zip: phrase, syllable and phoneme time boundaries (second) and labels in .txt format

The annotation_txt.zip folder annotations are represented as follows:

1. phrase_char: phrase-level time boundaries, labeled in Mandarin characters
2. phrase: phrase-level time boundaries, labeled in Mandarin pinyin
3. syllable: syllable-level time boundaries, labeled in Mandarin pinyin
4. phoneme: phoneme-level time boundaries, labeled in X-SAMPA

The boundaries (onset and offset) have been annotated hierarchically:

1. phrase (line)
2. syllable
3. phoneme

Annotation details:
Singing units in pinyin and X-SAMPA have been annotated to a jingju a cappella singing audio dataset.

Audio details:
The corresponding audio files are the a cappella singing arias recordings, which are stereo or mono, sampled at 44.1 kHz, and stored as .wav files. The .wav files are recorded by two institutes: those file names ending with 'qm' are recorded by C4DM, Queen Mary University of London; others file names ending with 'upf' or 'lon' are recorded by MTG-UPF. Additionally, another collection of 15 clean singing recordings is included in this dataset. They are extracted from the commercial recordings which originally contains karaoke accompaniment and mixed versions.

Additional details:
Annotation format, units, parsing code and other information please refer to: https://github.com/MTG/jingjuPhonemeAnnotation

License information:
Textgrid annotations are licensed under Creative Commons Attribution-NonCommercial 4.0 International License. Wav audio ending with 'upf' or 'lon' is licensed under Creative Commons Attribution-NonCommercial 4.0 International. For the license of .wav audio ending with 'qm' from C4DM Queen Mary University of London, please refer to this page http://isophonics.org/SingingVoiceDataset

---

`class mirdata.datasets.compmusic_jingju_acappella.Dataset(`*data_home=None*, *version='default'*`)`

The compmusic_jingju_acappella dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list

- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_multitracks`()

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

`load_phonemes`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.jingju_acapella.load_phonemes

`load_phrases`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.jingju_acapella.load_phrases

`load_syllable`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.jingju_acapella.load_syllable

`load_tracks`()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns
list – A list of track ids

`track_ids`

Return track ids

Returns
list – A list of track ids

`validate`(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

`class mirdata.datasets.compmusic_jingju_acappella.Track`(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

Jingju A Cappella Singing Track class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored. default=None If *None*, looks for the data in the default directory, *~/mir_datasets*

Variables

- **audio_path** (*str*) – local path where the audio is stored
- **phoneme_path** (*str*) – local path where the phoneme annotation is stored
- **phrase_char_path** (*str*) – local path where the lyric phrase annotation in chinese is stored
- **phrase_path** (*str*) – local path where the lyric phrase annotation in western characters is stored
- **syllable_path** (*str*) – local path where the syllable annotation is stored
- **work** (*str*) – string referring to the work where the track belongs
- **details** (*float*) – string referring to additional details about the track

Other Parameters

- **phoneme** (*EventData*) – phoneme annotation
- **phrase_char** (*LyricsData*) – lyric phrase annotation in chinese
- **phrase** (*LyricsData*) – lyric phrase annotation in western characters
- **syllable** (*EventData*) – syllable annotation

property audio: Tuple[numpy.ndarray, float] | None

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.compmusic_jingju_acappella.load_audio(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load Jingju A Cappella Singing audio file.

Parameters
fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.compmusic_jingju_acappella.load_phonemes(*fhandle: TextIO*) → *LyricData*

Load phonemes

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to a phoneme annotation file

Returns
LyricData – phoneme annotation

mirdata.datasets.compmusic_jingju_acappella.load_phrases(*fhandle: TextIO*) → *LyricData*

Load lyric phrases annotation

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to a lyric annotation file

Returns
LyricData – lyric phrase annotation

mirdata.datasets.compmusic_jingju_acappella.load_syllable(*fhandle: TextIO*) → *LyricData*

Load syllable

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to a syllable annotation file

Returns
LyricData – syllable annotation

2.5.14 compmusic_otmm_makam

OTMM Makam Recognition Dataset Loader

Dataset Info

NOTE: From mirdata v0.3.8 on, the only version available of this dataset is dlfm2016-fix1, which is basically the same as dlfm2016, but with a few fixes in some annotations. The original dlfm2016 version is still available in mirdata versions <=0.3.7. Note that from dlfm2016 to dlfm2016-fix1, no new recordings or annotation were added, only a few annotation files were fixed.

This dataset is designed to test makam recognition methodologies on Ottoman-Turkish makam music. It is composed of 50 recording from each of the 20 most common makams in CompMusic Project's Dunya Ottoman-Turkish Makam Music collection. Currently the dataset is the largest makam recognition dataset.

The recordings are selected from commercial recordings carefully such that they cover diverse musical forms, vocal/instrumentation settings and recording qualities (e.g. historical recordings vs. contemporary recordings). Each recording in the dataset is identified by an 16-character long unique identifier called MBID, hosted in MusicBrainz. The makam and the tonic of each recording is annotated in the file annotations.json.

The audio related data in the test dataset is organized by each makam in the folder data. Due to copyright reasons, we are unable to distribute the audio. Instead we provide the predominant melody of each recording, computed by a state-of-the-art predominant melody extraction algorithm optimized for OTMM culture. These features are saved as text files (with the paths data/[makam]/[mbid].pitch) of single column that contains the frequency values. The timestamps are removed to reduce the filesizes. The step size of the pitch track is 0.0029 seconds (an analysis window of 128 sample hop size of an mp3 with 44100 Hz sample rate), with which one can recompute the timestamps of samples.

Moreover the metadata of each recording is available in the repository, crawled from MusicBrainz using an open source tool developed by us. The metadata files are saved as data/[makam]/[mbid].json.

For reproducability purposes we note the version of all tools we have used to generate this dataset in the file algorithms.json (not integrated in the loader but present in the donwloaded dataset).

A complementary toolbox for this dataset is MORTY, which is a mode recogition and tonic identification toolbox. It can be used and optimized for any modal music culture. Further details are explained in the publication above.

`class mirdata.datasets.compmusic_otmm_makam.Dataset(data_home=None, version='default')`

The compmusic_otmm_makam dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

`choice_multitrack()`

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

`choice_track()`

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

`cite()`

Print the reference

`property default_path`

Get the default path for the dataset

Returns
str – Local path to the dataset

`download(partial_download=None, force_overwrite=False, cleanup=False, allow_invalid_checksum=False)`

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- `ValueError` – if invalid keys are passed to partial_download
- `IOError` – if a downloaded file's checksum is different from expected

`get_mtrack_splits()`

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- `AttributeError` – If this dataset does not have multitracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits(splits, seed=42, split_names=None)`

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_mb_tags`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.compmusic_otmm_makam.load_mb_tags

`load_multitracks`()

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

`load_pitch`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.compmusic_otmm_makam.load_pitch

`load_tracks`()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
`NotImplementedError` – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns
list – A list of track ids

track_ids

Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.compmusic_otmm_makam.Track(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

OTMM Makam Track class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored. default=None If *None*, looks for the data in the default directory, *~/mir_datasets*

Variables

- **pitch_path** (*str*) – local path where the pitch annotation is stored
- **mb_tags_path** (*str*) – local path where the MusicBrainz tags annotation is stored
- **makam** (*str*) – string referring to the makam represented in the track
- **tonic** (*float*) – tonic annotation
- **mbid** (*str*) – MusicBrainz ID of the track

Other Parameters

- **pitch** (*F0Data*) – pitch annotation
- **mb_tags** (*dict*) – dictionary containing the raw editorial track metadata from MusicBrainz

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.compmusic_otmm_makam.**load_mb_tags**(*fhandle: TextIO*) → dict

Load track metadata

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to musicbrainz metadata file

Returns
Dict – metadata of the track

mirdata.datasets.compmusic_otmm_makam.**load_pitch**(*fhandle: TextIO*) → *F0Data*

Load pitch

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to a pitch annotation file

Returns
F0Data – pitch annotation

2.5.15 compmusic_raga

CompMusic Raga Dataset Loader

Dataset Info

Rāga datasets from CompMusicomprise two sizable datasets, one for each music tradition, Carnatic and Hindustani. These datasets comprise full length audio recordings and their associated rāga labels. These two datasets can be used to develop and evaluate approaches for performing automatic rāga recognition in Indian art music.

These datasets are derived from the CompMusic corpora of Indian Art Music. Therefore, the dataset has been compiled at the Music Technology Group, by a group of researchers working on the computational analysis of Carnatic and Hindustani music within the framework of the ERC-funded CompMusic project.

Each recording is associated with a MBID. With the MBID other information can be obtained using the Dunya API or pycompmusic.

The Carnatic subset comprises 124 hours of audio recordings and editorial metadata that includes carefully curated and verified rāga labels. It contains 480 recordings belonging to 40 rāgas with 12 recordings per rāga.

The Hindustani subset comprises 116 hours of audio recordings and editorial metadata that includes carefully curated and verified rāga labels. It contains 300 recordings belonging to 30 rāgas with 10 recordings per rāga.

The dataset also includes features per each file: * Tonic: float indicating the recording tonic * Tonic fine tuned: float indicating the manually fine-tuned recording tonic * Predominant pitch: automatically-extracted predominant pitch time-series (timestamps and freq. values) * Post-processed pitch: automatically-extracted and post-processed predominant pitch time-series * Nyas segments: KNN-extracted segments of Nyas (start and end times provided) * Tani segments: KNN-extracted segments of Tanis (start and end times provided)

The dataset includes both txt files and json files that contain information about each audio recording in terms of its mbid, the path of the audio/feature files and the associated rāga identifier. Each rāga is assigned a unique identifier by Dunya, which is similar to the mbid in terms of purpose. A mapping of the rāga id to its transliterated name is also provided.

For more information about the dataset please refer to: https://compmusic.upf.edu/node/328

---

class mirdata.datasets.compmusic_raga.**Dataset**(*data_home=None*, *version='default'*)

The compmusic_raga dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

get_random_track_splits(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

get_track_splits()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- **AttributeError** – If this dataset does not have tracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

license()

Print the license

load_multitracks()

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
NotImplementedError – If the dataset does not support Multitracks

load_tracks()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

mtrack_ids

Return track ids

Returns
list – A list of track ids

track_ids

Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.compmusic_raga.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

CompMusic Raga Dataset class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored. default=None If *None*, looks for the data in the default directory, *~/mir_datasets*

Variables

- **audio_path** (*str*) – path to audio file
- **tonic_path** (*str*) – path to tonic annotation
- **tonic_fine_tuned_path** (*str*) – path to tonic fine-tuned annotation
- **pitch_path** (*str*) – path to pitch annotation
- **pitch_post_processed_path** (*str*) – path to processed pitch annotation
- **nyas_segments_path** (*str*) – path to nyas segments annotation
- **tani_segments_path** (*str*) – path to tani segments annotation

Other Parameters

- **tonic** (*float*) – tonic annotation
- **tonic_fine_tuned** (*float*) – tonic fine-tuned annotation
- **pitch** (*F0Data*) – pitch annotation

- **pitch_post_processed** (*F0Data*) – processed pitch annotation
- **nyas_segments** (*EventData*) – nyas segments annotation
- **tani_segments** (*EventData*) – tani segments annotation
- **recording** (*str*) – name of the recording
- **concert** (*str*) – name of the concert
- **artist** (*str*) – name of the artist
- **mbid** (*str*) – mbid of the recording
- **raga** (*str*) – raga in the recording
- **ragaid** (*str*) – id of the raga in the recording
- **tradition** (*str*) – tradition name (carnatic or hindustani)

`property audio`

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

`get_path`(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

`to_jams`()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

`mirdata.datasets.compmusic_raga.load_audio`(*audio_path*)

Load an audio file.

Parameters
audio_path (*str*) – path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

`mirdata.datasets.compmusic_raga.load_nyas_segments`(*fhandle*)

Load nyas segments

Parameters
fhandle (*str or file-like*) – Local path where the nyas segments annotation is stored.

Returns
EventData – segment annotation

mirdata.datasets.compmusic_raga.**load_pitch**(*fhandle*)

Load pitch

Parameters
fhandle (*str or file-like*) – Local path where the pitch annotation is stored.

Returns
F0Data – pitch annotation

mirdata.datasets.compmusic_raga.**load_tani_segments**(*fhandle*)

Load tani segments

Parameters
fhandle (*str or file-like*) – Local path where the tani segments annotation is stored.

Returns
EventData – segment annotation

mirdata.datasets.compmusic_raga.**load_tonic**(*fhandle*)

Load track absolute tonic

Parameters
fhandle (*str or file-like*) – Local path where the tonic path is stored.

Returns
int – Tonic annotation in Hz

2.5.16 dagstuhl_choirset

Dagstuhl ChoirSet Dataset Loader

Dataset Info

Dagstuhl ChoirSet (DCS) is a multitrack dataset of a cappella choral music. The dataset includes recordings of an amateur vocal ensemble performing two choir pieces in full choir and quartet settings (total duration 55min 30sec). The audio data was recorded during an MIR seminar at Schloss Dagstuhl using different close-up microphones to capture the individual singers' voices:

- Larynx microphone (LRX): contact microphone attached to the singer's throat.
- Dynamic microphone (DYN): handheld dynamic microphone.
- Headset microphone (HSM): microphone close to the singer's mouth.

LRX, DYN and HSM recordings are provided on the Track level. All tracks in the dataset have a LRX recording, while only a subset has DYN and HSM recordings.

In addition to the close-up microphone tracks, the dataset also provides the following recordings:

- Room microphone mixdown (STM): mixdown of the stereo room microphone.
- Room microphone left (STL): left channel of the stereo microphone.
- Room microphone right (STR): right channel of the stereo microphone.
- Room microphone mixdown with reverb (StereoReverb_STM): STM signal with artificial reverb.
- Piano left (SPL): left channel of the piano accompaniment.
- Piano right (SPR): right channel of the piano accompaniment.

All room microphone and piano recordings are provided on the Multitrack level. All multitracks have room microphone signals, while only a subset has piano recordings.

For more details, we refer to: Sebastian Rosenzweig (1), Helena Cuesta (2), Christof Weiß (1), Frank Scherbaum (3), Emilia Gómez (2,4), and Meinard Müller (1): Dagstuhl ChoirSet: A Multitrack Dataset for MIR Research on Choral Singing. Transactions of the International Society for Music Information Retrieval, 3(1), pp. 98–110, 2020. DOI: https://doi.org/10.5334/tismir.48

(1) International Audio Laboratories Erlangen, DE

(2) Music Technology Group, Universitat Pompeu Fabra, Barcelona, ES

(3) University of Potsdam, DE

(4) Joint Research Centre, European Commission, Seville, ES

---

class mirdata.datasets.dagstuhl_choirset.**Dataset**(*data_home=None*, *version='default'*)

The Dagstuhl ChoirSet dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns

str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- `ValueError` – if invalid keys are passed to partial_download
- `IOError` – if a downloaded file's checksum is different from expected

`get_mtrack_splits()`

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- `AttributeError` – If this dataset does not have multitracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits(splits, seed=42, split_names=None)`

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_random_track_splits(splits, seed=42, split_names=None)`

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_track_splits()`

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- **AttributeError** – If this dataset does not have tracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

license()
Print the license

load_audio(**args*, ***kwargs*)
Deprecated since version 0.3.4: Use mirdata.datasets.dagstuhl_choirset.load_audio

load_beat(**args*, ***kwargs*)
Deprecated since version 0.3.4: Use mirdata.datasets.dagstuhl_choirset.load_beat

load_f0(**args*, ***kwargs*)
Deprecated since version 0.3.4: Use mirdata.datasets.dagstuhl_choirset.load_f0

load_multitracks()
Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
NotImplementedError – If the dataset does not support Multitracks

load_score(**args*, ***kwargs*)
Deprecated since version 0.3.4: Use mirdata.datasets.dagstuhl_choirset.load_score

load_tracks()
Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

mtrack_ids
Return track ids

Returns
list – A list of track ids

track_ids
Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)
Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.dagstuhl_choirset.**MultiTrack**(*mtrack_id*, *data_home*, *dataset_name*, *index*, *track_class*, *metadata*)

Dagstuhl ChoirSet multitrack class

Parameters

- **mtrack_id** (*str*) – multitrack id
- **data_home** (*str*) – Local path where the dataset is stored. If *None*, looks for the data in the default directory, *~/mir_datasets/dagstuhl_choirset*

Variables

- **audio_stm_path** (*str*) – path to room mic (mono mixdown) audio file
- **audio_str_path** (*str*) – path to room mic (right channel) audio file
- **audio_stl_path** (*str*) – path to room mic (left channel) audio file
- **audio_rev_path** (*str*) – path to room mic with artifical reverb (mono mixdown) audio file
- **audio_spl_path** (*str*) – path to piano accompaniment (left channel) audio file
- **audio_spr_path** (*str*) – path to piano accompaniement (right channel) audio file
- **beat_path** (*str*) – path to beat annotation file

Other Parameters

- **beat** (*annotations.BeatData*) – Beat annotation
- **notes** (*annotations.NoteData*) – Note annotation
- **multif0** (*annotations.MultiF0Data*) – Aggregate of f0 annotations for tracks

property audio_rev: Tuple[numpy.ndarray, float] | None

The audio for the room mic with artifical reverb (mono mixdown)

Returns

- np.ndarray - audio signal
- float - sample rate

property audio_spl: Tuple[numpy.ndarray, float] | None

The audio for the piano accompaniment DI (left channel)

Returns

- np.ndarray - audio signal
- float - sample rate

property audio_spr: Tuple[numpy.ndarray, float] | None

The audio for the piano accompaniment DI (right channel)

Returns

- np.ndarray - audio signal
- float - sample rate

property audio_stl: Tuple[numpy.ndarray, float] | None

The audio for the room mic (left channel)

Returns

- np.ndarray - audio signal

- float - sample rate

property audio_stm: Tuple[numpy.ndarray, float] | None

The audio for the room mic (mono mixdown)

Returns

- np.ndarray - audio signal
- float - sample rate

property audio_str: Tuple[numpy.ndarray, float] | None

The audio for the room mic (right channel)

Returns

- np.ndarray - audio signal
- float - sample rate

get_mix()

Create a linear mixture given a subset of tracks.

Parameters
track_keys (*list*) – list of track keys to mix together

Returns
np.ndarray – mixture audio with shape (n_samples, n_channels)

get_path(*key*)

Get absolute path to multitrack audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

get_random_target(*n_tracks=None*, *min_weight=0.3*, *max_weight=1.0*)

Get a random target by combining a random selection of tracks with random weights

Parameters

- **n_tracks** (*int or None*) – number of tracks to randomly mix. If None, uses all tracks
- **min_weight** (*float*) – minimum possible weight when mixing
- **max_weight** (*float*) – maximum possible weight when mixing

Returns

- np.ndarray - mixture audio with shape (n_samples, n_channels)
- list - list of keys of included tracks
- list - list of weights used to mix tracks

get_target(*track_keys*, *weights=None*, *average=True*, *enforce_length=True*)

Get target which is a linear mixture of tracks

Parameters

- **track_keys** (*list*) – list of track keys to mix together
- **weights** (*list or None*) – list of positive scalars to be used in the average

- **average** (*bool*) – if True, computes a weighted average of the tracks if False, computes a weighted sum of the tracks
- **enforce_length** (*bool*) – If True, raises ValueError if the tracks are not the same length. If False, pads audio with zeros to match the length of the longest track

Returns
np.ndarray – target audio with shape (n_channels, n_samples)

Raises
ValueError – if sample rates of the tracks are not equal if enforce_length=True and lengths are not equal

`to_jams()`

Jams: the track's data in jams format

`class mirdata.datasets.dagstuhl_choirset.Track(track_id, data_home, dataset_name, index, metadata)`

Dagstuhl ChoirSet Track class

Parameters
track_id (*str*) – track id of the track

Variables

- **audio_dyn_path** (*str*) – dynamic microphone audio path
- **audio_hsm_path** (*str*) – headset microphone audio path
- **audio_lrx_path** (*str*) – larynx microphone audio path
- **f0_crepe_dyn_path** (*str*) – crepe f0 annotation for dynamic microphone path
- **f0_crepe_hsm_path** (*str*) – crepe f0 annotation for headset microphone path
- **f0_crepe_lrx_path** (*str*) – crepe f0 annotation for larynx microphone path
- **f0_pyin_dyn_path** (*str*) – pyin f0 annotation for dynamic microphone path
- **f0_pyin_hsm_path** (*str*) – pyin f0 annotation for headset microphone path
- **f0_pyin_lrx_path** (*str*) – pyin f0 annotation for larynx microphone path
- **f0_manual_lrx_path** (*str*) – manual f0 annotation for larynx microphone path
- **score_path** (*str*) – score annotation path

Other Parameters

- **f0_crepe_dyn** (*F0Data*) – algorithm-labeled (crepe) f0 annotations for dynamic microphone
- **f0_crepe_hsn** (*F0Data*) – algorithm-labeled (crepe) f0 annotations for headset microphone
- **f0_crepe_lrx** (*F0Data*) – algorithm-labeled (crepe) f0 annotations for larynx microphone
- **f0_pyin_dyn** (*F0Data*) – algorithm-labeled (pyin) f0 annotations for dynamic microphone
- **f0_pyin_hsn** (*F0Data*) – algorithm-labeled (pyin) f0 annotations for headset microphone
- **f0_pyin_lrx** (*F0Data*) – algorithm-labeled (pyin) f0 annotations for larynx microphone
- **f0_manual_lrx** (*F0Data*) – manually labeled f0 annotations for larynx microphone
- **score** (*NoteData*) – time-aligned score representation

property audio_dyn: Tuple[numpy.ndarray, float] | None

The audio for the track's dynamic microphone (if available)

Returns

- np.ndarray - audio signal
- float - sample rate

property audio_hsm: Tuple[numpy.ndarray, float] | None

The audio for the track's headset microphone (if available)

Returns

- np.ndarray - audio signal
- float - sample rate

property audio_lrx: Tuple[numpy.ndarray, float] | None

The audio for the track's larynx microphone (if available)

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Jams: the track's data in jams format

mirdata.datasets.dagstuhl_choirset.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a Dagstuhl ChoirSet audio file.

Parameters
audio_path (*str*) – path pointing to an audio file

Returns

- np.ndarray - the audio signal
- float - The sample rate of the audio file

mirdata.datasets.dagstuhl_choirset.**load_beat**(*fhandle: TextIO*) → *BeatData*

Load a Dagstuhl ChoirSet beat annotation.

Parameters
fhandle (*str or file-like*) – File-like object or path to beat annotation file

Returns
BeatData Object - the beat annotation

mirdata.datasets.dagstuhl_choirset.**load_f0**(*fhandle: TextIO*) → *F0Data*

Load a Dagstuhl ChoirSet F0-trajectory.

Parameters
fhandle (*str or file-like*) – File-like object or path to F0 file

Returns
F0Data Object - the F0-trajectory

mirdata.datasets.dagstuhl_choirset.**load_score**(*fhandle: TextIO*) → *NoteData*

Load a Dagstuhl ChoirSet time-aligned score representation.

Parameters
fhandle (*str or file-like*) – File-like object or path to score representation file

Returns
NoteData Object - the time-aligned score representation

2.5.17 dali

DALI Dataset Loader

Dataset Info

DALI contains 5358 audio files with their time-aligned vocal melody. It also contains time-aligned lyrics at four levels of granularity: notes, words, lines, and paragraphs.

For each song, DALI also provides additional metadata: genre, language, musician, album covers, or links to video clips.

For more details, please visit: https://github.com/gabolsgabs/DALI

---

class mirdata.datasets.dali.**Dataset**(*data_home=None*, *version='default'*)

The dali dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

`cite()`

Print the reference

`property default_path`

Get the default path for the dataset

Returns
str – Local path to the dataset

`download(partial_download=None, force_overwrite=False, cleanup=False, allow_invalid_checksum=False)`

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- `ValueError` – if invalid keys are passed to partial_download
- `IOError` – if a downloaded file's checksum is different from expected

`get_mtrack_splits()`

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- `AttributeError` – If this dataset does not have multitracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits(splits, seed=42, split_names=None)`

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_random_track_splits(splits, seed=42, split_names=None)`

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_track_splits()`

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license()`

Print the license

`load_annotations_class(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.dali.load_annotations_class

`load_annotations_granularity(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.dali.load_annotations_granularity

`load_audio(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.dali.load_audio

`load_multitracks()`

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

`load_tracks()`

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
`NotImplementedError` – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns
list – A list of track ids

`track_ids`

Return track ids

Returns
list – A list of track ids

`validate`(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

`class mirdata.datasets.dali.Track`(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

DALI melody Track class

Parameters
track_id (*str*) – track id of the track

Variables

- **album** (*str*) – the track's album
- **annotation_path** (*str*) – path to the track's annotation file
- **artist** (*str*) – the track's artist
- **audio_path** (*str*) – path to the track's audio file
- **audio_url** (*str*) – youtube ID
- **dataset_version** (*int*) – dataset annotation version
- **ground_truth** (*bool*) – True if the annotation is verified
- **language** (*str*) – sung language
- **release_date** (*str*) – year the track was released
- **scores_manual** (*int*) – manual score annotations
- **scores_ncc** (*float*) – ncc score annotations
- **title** (*str*) – the track's title
- **track_id** (*str*) – the unique track id
- **url_working** (*bool*) – True if the youtube url was valid

Other Parameters

- **notes** (*NoteData*) – vocal notes
- **words** (*LyricData*) – word-level lyrics
- **lines** (*LyricData*) – line-level lyrics
- **paragraphs** (*LyricData*) – paragraph-level lyrics
- **annotation-object** (*DALI.Annotations*) – DALI annotation object

`property audio: Tuple[numpy.ndarray, float] | None`

The track's audio

Returns

- np.ndarray - audio signal

- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.dali.**load_annotations_class**(*annotations_path*)

Load full annotations into the DALI class object

Parameters
annotations_path (*str*) – path to a DALI annotation file

Returns
DALI.annotations – DALI annotations object

mirdata.datasets.dali.**load_annotations_granularity**(*annotations_path*, *granularity*)

Load annotations at the specified level of granularity

Parameters

- **annotations_path** (*str*) – path to a DALI annotation file
- **granularity** (*str*) – one of 'notes', 'words', 'lines', 'paragraphs'

Returns
NoteData for granularity='notes' or LyricData otherwise

mirdata.datasets.dali.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float] | None

Load a DALI audio file.

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to an audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

2.5.18 da_tacos

Da-TACOS Dataset Loader

Dataset Info

Da-TACOS: a dataset for cover song identification and understanding. It contains two subsets, namely the benchmark subset (for benchmarking cover song identification systems) and the cover analysis subset (for analyzing the links among cover songs), with pre-extracted features and metadata for 15,000 and 10,000 songs, respectively. The annotations included in the metadata are obtained with the API of SecondHandSongs.com. All audio files we use to extract features are encoded in MP3 format and their sample rate is 44.1 kHz. Da-TACOS does not contain any audio files. For the

results of our analyses on modifiable musical characteristics using the cover analysis subset and our initial benchmarking of 7 state-of-the-art cover song identification algorithms on the benchmark subset, you can look at our publication.

For organizing the data, we use the structure of SecondHandSongs where each song is called a 'performance', and each clique (cover group) is called a 'work'. Based on this, the file names of the songs are their unique performance IDs (PID, e.g. P_22), and their labels with respect to their cliques are their work IDs (WID, e.g. W_14).

Metadata for each song includes:

- performance title
- performance artist
- work title
- work artist
- release year
- SecondHandSongs.com performance ID
- SecondHandSongs.com work ID
- whether the song is instrumental or not

In addition, we matched the original metadata with MusicBrainz to obtain MusicBrainz ID (MBID), song length and genre/style tags. We would like to note that MusicBrainz related information is not available for all the songs in Da-TACOS, and since we used just our metadata for matching, we include all possible MBIDs for a particular songs.

For facilitating reproducibility in cover song identification (CSI) research, we propose a framework for feature extraction and benchmarking in our supplementary repository: acoss. The feature extraction component is designed to help CSI researchers to find the most commonly used features for CSI in a single address. The parameter values we used to extract the features in Da-TACOS are shared in the same repository. Moreover, the benchmarking component includes our implementations of 7 state-of-the-art CSI systems. We provide the performance results of an initial benchmarking of those 7 systems on the benchmark subset of Da-TACOS. We encourage other CSI researchers to contribute to acoss with implementing their favorite feature extraction algorithms and their CSI systems to build up a knowledge base where CSI research can reach larger audiences.

Pre-extracted features:

The list of features included in Da-TACOS can be seen below. All the features are extracted with acoss repository that uses open-source feature extraction libraries such as Essentia, LibROSA, and Madmom.

To facilitate the use of the dataset, we provide two options regarding the file structure.

1. In da-tacos_benchmark_subset_single_files and da-tacos_coveranalysis_subset_single_files folders, we organize the data based on their respective cliques, and one file contains all the features for that particular song.

```
{
    "chroma_cens": numpy.ndarray,
    "crema": numpy.ndarray,
    "hpcp": numpy.ndarray,
    "key_extractor": {
        "key": numpy.str_,
        "scale": numpy.str_,_
        "strength": numpy.float64
    },
    "madmom_features": {
        "novfn": numpy.ndarray,
        "onsets": numpy.ndarray,
        "snovfn": numpy.ndarray,
```

(continues on next page)

(continued from previous page)

```
        "tempos": numpy.ndarray
    }
    "mfcc_htk": numpy.ndarray,
    "tags": list of (numpy.str_, numpy.str_)
    "label": numpy.str_,
    "track_id": numpy.str_
}
```

2. In da-tacos_benchmark_subset_FEATURE and da-tacos_coveranalysis_subset_FEATURE folders, the data is organized based on their cliques as well, but each of these folders contain only one feature per song. For instance, if you want to test your system that uses HPCP features, you can download da-tacos_benchmark_subset_hpcp to access the pre-computed HPCP features. An example for the contents in those files can be seen below:

```
{
    "hpcp": numpy.ndarray,
    "label": numpy.str_,
    "track_id": numpy.str_
}
```

---

class mirdata.datasets.da_tacos.**Dataset**(*data_home=None*, *version='default'*)

The Da-TACOS dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

benchmark_tracks()

Load from Da-TACOS dataset the benchmark subset tracks.

Returns

dict – {*track_id*: track data}

choice_multitrack()

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

`cite()`

Print the reference

`coveranalysis_tracks()`

Load from Da-TACOS dataset the coveranalysis subset tracks.

Returns
dict – {*track_id*: track data}

`property default_path`

Get the default path for the dataset

Returns
str – Local path to the dataset

`download(partial_download=None, force_overwrite=False, cleanup=False, allow_invalid_checksum=False)`

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- `ValueError` – if invalid keys are passed to partial_download
- `IOError` – if a downloaded file's checksum is different from expected

`filter_index(search_key)`

Load from Da-TACOS genre dataset the indexes that match with search_key.

Parameters
search_key (*str*) – regex to match with folds, mbid or genres

Returns
dict – {*track_id*: track data}

`get_mtrack_splits()`

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- `AttributeError` – If this dataset does not have multitracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits(splits, seed=42, split_names=None)`

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_cens`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.da_tacos.load_cens

`load_crema`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.da_tacos.load_crema

`load_hpcp`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.da_tacos.load_hpcp

`load_key`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.da_tacos.load_key

`load_madmom`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.da_tacos.load_madmom

`load_mfcc`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.da_tacos.load_mfcc

load_multitracks()

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
NotImplementedError – If the dataset does not support Multitracks

load_tags(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.da_tacos.load_tags

load_tracks()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

mtrack_ids

Return track ids

Returns
list – A list of track ids

track_ids

Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.da_tacos.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

da_tacos track class

Parameters
track_id (*str*) – track id of the track

Variables

- **subset** (*str*) – subset which the track belongs to
- **work_id** (*str*) – id of work's original track
- **label** (*str*) – alias of work_id
- **performance_id** (*str*) – id of cover track
- **cens_path** (*str*) – cens annotation path
- **crema_path** (*str*) – crema annotation path

- **hpcp_path** (*str*) – hpcp annotation path
- **key_path** (*str*) – key annotation path
- **madmom_path** (*str*) – madmom annotation path
- **mfcc_path** (*str*) – mfcc annotation path
- **tags_path** (*str*) – tags annotation path

Properties:

work_title (str): title of the work work_artist (str): original artist of the work performance_title (str): title of the performance performance_artist (str): artist of the performance release_year (str): release year is_instrumental (bool): True if the track is instrumental performance_artist_mbid (str): musicbrainz id of the performance artist mb_performances (dict): musicbrainz ids of performances

Other Parameters

- **cens** (*np.ndarray*) – chroma-cens features
- **hpcp** (*np.ndarray*) – hpcp features
- **key** (*dict*) – key data, with keys 'key', 'scale', and 'strength'
- **madmom** (*dict*) – dictionary of madmom analysis features
- **mfcc** (*np.ndarray*) – mfcc features
- **tags** (*list*) – list of tags

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams() → jams.JAMS

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.da_tacos.**load_cens**(*fhandle: BinaryIO*)

Load Da-TACOS cens features from a file

Parameters
fhandle (*str or file-like*) – File-like object or path to chroma-cens file

Returns
np.ndarray – cens features

mirdata.datasets.da_tacos.**load_crema**(*fhandle: BinaryIO*)

Load Da-TACOS crema features from a file

Parameters
fhandle (*str or file-like*) – File-like object or path to crema file

Returns
np.ndarray – crema features

mirdata.datasets.da_tacos.**load_hpcp**(*fhandle: BinaryIO*)

Load Da-TACOS hpcp features from a file

Parameters
fhandle (*str or file-like*) – File-like object or path to hpcp file

Returns
np.ndarray – hpcp features

mirdata.datasets.da_tacos.**load_key**(*fhandle: BinaryIO*)

Load Da-TACOS key features from a file.

Parameters
fhandle (*str or file-like*) – File-like object or path to key file

Returns
dict – key, mode and confidence

Examples

{'key': 'C', 'scale': 'major', 'strength': 0.8449875116348267}

---

mirdata.datasets.da_tacos.**load_madmom**(*fhandle: BinaryIO*)

Load Da-TACOS madmom features from a file

Parameters
fhandle (*str or file-like*) – File-like object or path to madmom file

Returns
dict – madmom features, with keys 'novfn', 'onsets', 'snovfn', 'tempos

mirdata.datasets.da_tacos.**load_mfcc**(*fhandle: BinaryIO*)

Load Da-TACOS mfcc from a file

Parameters
fhandle (*str or file-like*) – File-like object or path to mfcc file

Returns
np.ndarray – array of mfccs over time

mirdata.datasets.da_tacos.**load_tags**(*fhandle: BinaryIO*)

Load Da-TACOS tags from a file

Parameters
fhandle (*str or file-like*) – File-like object or path to tags file

Returns
list – tags, in the form [(tag, confidence), ...]

Example

[('rock', '0.127'), ('pop', '0.014'), ...]

2.5.19 egfxset

EGFxSet Dataset Loader

Dataset Info

EGFxSet (Electric Guitar Effects dataset) features recordings for all clean tones in a 22-fret Stratocaster, recorded with 5 different pickup configurations, also processed through 12 popular guitar effects. Our dataset was recorded in real hardware, making it relevant for music information retrieval tasks on real music. We also include annotations for parameter settings of the effects we used.

EGFxSet is a dataset of 8,970 audio files with a 5-second duration each, summing a total time of - 12 hours and 28 minutes -.

All possible 138 notes of a standard tuning 22 frets guitar were recorded in each one of the 5 pickup configurations, giving a total of 690 clean tone audio files (58 min).

The 690 clean audio (58 min) files were processed through 12 different audio effects employing actual guitar gear (no VST emulations were used), summing a total of 8,280 processed audio files (11 hours 30 min).

The effects employed were divided into four categories, and each category comprised three different effects. Sometimes there were employed more than one effect from a same guitar equipment.

Categories, Models and Effects:

Distortion:

Boss BD-2:
Blues Driver

Ibanez Minitube Screamer:
Tube Screamer

ProCo RAT2:
Distortion

Modulation:

Boss CE-3:
Chorus

MXR Phase 45:
Phaser

Mooer E-Lady:
Flanger

Delays:

Line6 DL-4:
Digital Delay, Tape Echo, Sweep Echo

Reverb:

Orange CR-60 Combo Amplifier:
Plate Reverb, Hall Reverb, Spring Reverb

Annotations are labeled by a trained electric guitar musician. For each tone, we provide:

- Guitar string number
- Fret number
- Guitar pickup configuration

- Effect name
- Effect type
- Hardware modes
- Knob names
- Knob types
- Knob settings

The dataset website is: https://egfxset.github.io/

The data can be accessed here: https://zenodo.org/record/7044411#.YxKdSWzMKEI

An ISMIR extended abstract was presented in 2022: https://ismir2022.ismir.net/program/lbd/

This dataset was conceived during Iran Roman's "Deep Learning for Music Information Retrieval" course imparted in the postgraduate studies in music technology at the UNAM (Universidad Nacional Autónoma de México). The result is a combined effort between two UNAM postgraduate students (Hegel Pedroza and Gerardo Meza) and Iran Roman(NYU).

---

class mirdata.datasets.egfxset.**Dataset**(*data_home=None*, *version='default'*)

The EGFxSet dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

`download`(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- `ValueError` – if invalid keys are passed to partial_download
- `IOError` – if a downloaded file's checksum is different from expected

`get_mtrack_splits`()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- `AttributeError` – If this dataset does not have multitracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits`(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_track_splits()`

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- **AttributeError** – If this dataset does not have tracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license()`

Print the license

`load_multitracks()`

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
NotImplementedError – If the dataset does not support Multitracks

`load_tracks()`

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns
list – A list of track ids

`track_ids`

Return track ids

Returns
list – A list of track ids

`validate(verbose=True)`

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

`class mirdata.datasets.egfxset.Track(track_id, data_home, dataset_name, index, metadata)`

EGFxSet Track class

Parameters
track_id (*str*) – track id of the track

Variables

- **audio_path** (*str*) – path to the track's audio file
- **stringfret_tuple** (*list*) – an array with the tuple of the note recorded
- **pickup_configuration** (*string*) – the pickup used in the recording
- **effect** (*str*) – the effect recorded
- **model** (*str*) – the model of the hardware used
- **effect_type** (*str) the type of effect used (distortion, modulation, delay or reverb*) –
- **knob_names** (*list*) – an array with the knob names of the effect used or "None" when the recording is a clean effect sound
- **knob_type** (*list*) – an array with the type of knobs of the effect used or "None" when the recording is a clean effect sound
- **setting** (*list*) – the setting of the effect recorded or "None" when the recording is a clean effect sound

Other Parameters

- **note_name** (*ndarray*) – a list with the note name annotation of the audio file (e.g. "Ab5", "C6", etc.)
- **midinote** (*NoteData*) – the midinote annotation of the audio file

property audio: Tuple[numpy.ndarray, float] | None

Solo guitar audio (mono)

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.egfxset.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load EGFxSet guitar audio

Parameters
fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - audio signal
- float - sample rate

2.5.20 filosax

Filosax Dataset Loader

Dataset Info

The Filosax dataset was conceived, curated and compiled by Dave Foster (a PhD student on the AIM programme at QMUL) and his supervisor Simon Dixon (C4DM @ QMUL). The dataset is a collection of 48 multitrack jazz recordings, where each piece has 8 corresponding audio files:

1) The original Aebersold backing track (stereo)
2) Bass_Drums, a mono file of a mix of bass and drums
3) Piano_Drums, a mono file of a mix of piano and drums
4) Participant 1 Sax, a mono file of solo saxophone
5) Participant 2 Sax, a mono file of solo saxophone
6) Participant 3 Sax, a mono file of solo saxophone
7) Participant 4 Sax, a mono file of solo saxophone
8) Participant 5 Sax, a mono file of solo saxophone

Each piece is ~6mins long, so each of the 8 stems contains ~5hours of audio

For each piece, there is a corresponding .jams file containing piece-level annotations:

1) Beat annotation for the start of each bar and any mid-bar chord change
2) Chord annotation for each bar, and mid-bar chord change
3) **Section annotation for when the solo changes between the 3 categories:**
    a) head (melody)
    b) written solo (interpretation of transcribed solo)
    c) improvised solo

For each Sax recording (5 per piece), there is a corresponding .json file containing note annotations (see Note object).

The Participant folders also contain MIDI files of the transcriptions (frame level and score level) as well as a PDF and MusicXML of the typeset solo.

The dataset comes in 2 flavours: full (all 48 tracks and 5 sax players) and lite (5 tracks and 2 sax players). Both flavours can be used with or without the backing tracks (which need to be purchased online). Hence, when opening the dataset, use one of 4 versions: 'full', 'full_sax', 'lite', 'lite_sax'.

`class mirdata.datasets.filosax.Dataset(`*data_home=None*, *version='default'*`)`

The Filosax dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –

- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns

str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits`(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_multitracks`()

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

`load_tracks`()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns
list – A list of track ids

`track_ids`

Return track ids

Returns
list – A list of track ids

`validate`(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

`class mirdata.datasets.filosax.MultiTrack`(*mtrack_id*, *data_home*, *dataset_name*, *index*, *track_class*, *metadata*)

Filosax multitrack class

Parameters

- **mtrack_id** (*str*) – multitrack id
- **data_home** (*str*) – Local path where the dataset is stored. If *None*, looks for the data in the default directory, *~/mir_datasets/Filosax*

Variables

- **mtrack_id** (*str*) – track id
- **tracks** (*dict*) – {track_id: Track}
- **track_audio_property** (*str*) – the name of the attribute of Track which returns the audio to be mixed
- **name** (*str*) – the name of the tune
- **duration** (*float*) – the duration, in seconds
- **beats** (*list, Observation*) – the time and beat numbers of bars and chord changes
- **chords** (*list, Observation*) – the time of chord changes
- **segments** (*list, Observation*) – the time of segment changes
- **bass_drums** (Track) – the associated bass/drums track
- **piano_drums** (Track) – the associated piano/drums track
- **sax** (*list,* Track) – a list of associated sax tracks

Other Parameters
annotation (*jams.JAMS*) – a .jams file containing the annotations

`annotation`

.jams file

Type
output type

`property bass_drums`

The associated bass/drums track

Returns

- Track

`property beats`

The times of downbeats and chord changes

Returns

- (SortedKeyList, Observation) - timestamp, duration (seconds), beat

`property chords`

The times and values of chord changes

Returns

- (SortedKeyList, Observation) - timestamp, duration (seconds), chord symbol

`property duration`

The track's duration

Returns

- float - track duration (in seconds)

`get_mix()`

Create a linear mixture given a subset of tracks.

Parameters
track_keys (*list*) – list of track keys to mix together

Returns
np.ndarray – mixture audio with shape (n_samples, n_channels)

`get_path(key)`

Get absolute path to multitrack audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

`get_random_target(n_tracks=None, min_weight=0.3, max_weight=1.0)`

Get a random target by combining a random selection of tracks with random weights

Parameters

- **n_tracks** (*int or None*) – number of tracks to randomly mix. If None, uses all tracks
- **min_weight** (*float*) – minimum possible weight when mixing
- **max_weight** (*float*) – maximum possible weight when mixing

Returns

- np.ndarray - mixture audio with shape (n_samples, n_channels)
- list - list of keys of included tracks
- list - list of weights used to mix tracks

`get_target`(*track_keys*, *weights=None*, *average=True*, *enforce_length=True*)

Get target which is a linear mixture of tracks

Parameters

- **track_keys** (*list*) – list of track keys to mix together
- **weights** (*list or None*) – list of positive scalars to be used in the average
- **average** (*bool*) – if True, computes a weighted average of the tracks if False, computes a weighted sum of the tracks
- **enforce_length** (*bool*) – If True, raises ValueError if the tracks are not the same length. If False, pads audio with zeros to match the length of the longest track

Returns

np.ndarray – target audio with shape (n_channels, n_samples)

Raises

`ValueError` – if sample rates of the tracks are not equal if enforce_length=True and lengths are not equal

`property name`

The track's name

Returns

- str - track name

`property piano_drums`

The associated piano/drums track

Returns

- Track

`property sax`

The associated sax tracks (1-5)

Returns

- (list, Track)

`property segments`

The times of segment changes (values are 'head', 'written solo', 'improvised solo')

Returns

- (SortedKeyList, Observation) - timestamp, duration (seconds), beat

`to_jams`()

Jams: the track's data in jams format

`class mirdata.datasets.filosax.Note`(*input_dict*)

Filosax Note class - dictionary wrapper to give dot properties

Parameters

input_dict (*dict*) – dictionary of attributes

Variables

- **a_start_time** (*float*) – the time stamp of the note start, in seconds
- **a_end_time** (*float*) – the time stamp of the note end, in seconds
- **a_duration** (*float*) – the duration of the note, in seconds
- **a_onset_time** (*float*) – the onset time (compared to a_start_time) (filosax_full only, 0.0 otherwise)
- **midi_pitch** (*int*) – the quantised midi pitch
- **crochet_num** (*int*) – the number of sub-divisions which define a crochet (always 24)
- **musician** (*int*) – the participant ID
- **bar_num** (*int*) – the bar number of the start of the note
- **s_start_time** (*float*) – the time stamp of the score note start, in seconds
- **s_duration** (*float*) – the duration of the score note, in seconds
- **s_end_time** (*float*) – the time stamp of the score note end, in seconds
- **s_rhythmic_duration** (*int*) – the duration of the score note (compared to crochet_num)
- **s_rhythmic_position** (*int*) – the position in the bar of the score note start (compared to crochet_num)
- **tempo** (*float*) – the tempo at the start of the note, in beats per minute
- **bar_type** (*int*) – the section annotation where 0 = head, 1 = written solo, 2 = improvised solo
- **is_grace** (*bool*) – is the note a grace note, associated with the following note
- **chord_changes** (*dict*) – the chords, where the key is the rhythmic position of the chord (using crochet_num, relative to s_rhythmic_position) and the value a JAMS chord annotation (An additional chord is added in the case of a quaver at the end of the bar, followed by a rest on the downbeat)
- **num_chord_changes** (*int*) – the number of chords which accompany the note (usually 1, sometimes >1 for long notes)
- **main_chord_num** (*int*) – usually 0, sometimes 1 in the quaver case described above
- **scale_changes** (*list, int*) – the degree of the chromatic scale when midi_pitch is compared to chord_root
- **loudness_max_val** (*float*) – the value (db) of the maximum loudness
- **loudness_max_time** (*float*) – the time (seconds) of the maximum loudness (compared to a_start_time)
- **loudness_curve** (*list, float*) – the inter-note loudness values, 1 per millisecond
- **pitch_average_val** (*float*) – the value (midi) of the average pitch and
- **pitch_average_time** (*float*) – the time (seconds) of the average pitch (compared to a_start_time)
- **pitch_curve** (*list, float*) – the inter-note pitch values, 1 per millisecond
- **pitch_vib_freq** (*float*) – the vibrato frequency (Hz), 0.0 if no vibrato detected
- **pitch_vib_ext** (*float*) – the vibrato extent (midi), 0.0 if no vibrato detected

- **spec_cent** (*float*) – the spectral centroid value at the time of the maximum loudness
- **spec_flux** (*float*) – the spectral flux value at the time of the maximum loudness
- **spec_cent_curve** (*list, float*) – the inter-note spectral centroid values, 1 per millisecond
- **spec_flux_curve** (*list, float*) – the inter-note spectral flux values, 1 per millisecond
- **seq_len** (*int*) – the length of the phrase in which the note falls (filosax_full only, -1 otherwise)
- **seq_num** (*int*) – the note position in the phrase (filosax_full only, -1 otherwise)

class mirdata.datasets.filosax.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

Filosax track class

Parameters
track_id (*str*) – track id of the track

Variables

- **audio_path** (*str*) – path to audio file
- **annotation_path** (*str*) – path to annotation file
- **midi_path** (*str*) – path to MIDI file
- **musicXML_path** (*str*) – path to musicXML file
- **pdf_path** (*str*) – path to PDF file

Other Parameters
notes (*list, Note*) – an ordered list of Note objects

property audio: Tuple[numpy.ndarray, float] | None

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

notes

The track's note list - only for Sax files

Returns

- [Note] - ordered list of Note objects (empty if Backing file)

mirdata.datasets.filosax.**load_annotation**(*fhandle: TextIO*) → List[*Note*]

Load a Filosax annotation file.

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to an audio file

Returns
** (list, Note)* – an ordered list of Note objects

mirdata.datasets.filosax.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a Filosax audio file.

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to an audio file

Returns

- np.ndarray - the audio signal
- float - The sample rate of the audio file

2.5.21 four_way_tabla

Four-Way Tabla Stroke Transcription and Classification Loader

---

Dataset Info

The Four-Way Tabla Dataset includes audio recordings of tabla solo with onset annotations for particular strokes types. This dataset was published in 2021 in the context of ISMIR2021 (Online), and may be used for tasks related to tabla analysis, including problems such as onset detection and stroke classification.

Total audio samples: We do have a total of 226 samples for training and 10 for testing. Each audio has an approximate duration of 1 minute.

Audio specifications:

- Sampling frequency: 44.1 kHz
- Bit-depth: 16 bit
- Audio format: .wav

Dataset usage: This dataset may be used for the data-driven research of tabla stroke transcription and identification. In this dataset, four important tabla characteristic strokes are considered.

Dataset structure: The dataset is split in two subsets, containing training and testing samples. Within each subset, there is a folder containing the audios, and another folder containing the onset annotations. The onset annotations are organized in a folder per each stroke type: b, d, rb, rt. Therefore, the paths to onsets would look like:

```
train/onsets/<StrokeType>/<ID>.onsets
```

The dataset is made available by CompMusic under a Creative Commons Attribution 3.0 Unported (CC BY 3.0) License.

---

class mirdata.datasets.four_way_tabla.**Dataset**(*data_home=None*, *version='default'*)

The Four-Way Tabla dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format

- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits`(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_multitracks`()

Load all multitracks in the dataset

Returns

dict – {*mtrack_id*: multitrack data}

Raises

`NotImplementedError` – If the dataset does not support Multitracks

`load_tracks`()

Load all tracks in the dataset

Returns

dict – {*track_id*: track data}

Raises

`NotImplementedError` – If the dataset does not support Tracks

mtrack_ids

Return track ids

Returns
list – A list of track ids

track_ids

Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.four_way_tabla.Track(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

Four-Way Tabla track class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored.

Variables

- **track_id** (*str*) – track id
- **audio_path** (*str*) – audio path
- **onsets_b_path** (*str*) – path to B onsets
- **onsets_d_path** (*str*) – path to D onsets
- **onsets_rb_path** (*str*) – path to RB onsets
- **onsets_rt_path** (*str*) – path to RT onsets

property audio: Tuple[numpy.ndarray, float] | None

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

property onsets_b: *BeatData* | None

Onsets for stroke B

Returns

- annotations.BeatData - onsets annotation

property onsets_d: *BeatData* | None

Onsets for stroke D

Returns

- annotations.BeatData - onsets annotation

property onsets_rb: *BeatData* | None

Onsets for stroke RB

Returns

- annotations.BeatData - onsets annotation

property onsets_rt: *BeatData* | None

Onsets for stroke RT

Returns

- annotations.BeatData - onsets annotation

to_jams()

Get the track's data in jams format

Returns

jams.JAMS – the track's data in jams format

mirdata.datasets.four_way_tabla.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a Mridangam Stroke Dataset audio file.

Parameters

fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.four_way_tabla.**load_onsets**(*fhandle*)

Load stroke onsets

Parameters

fhandle (*str or file-like*) – Local path where the pitch annotation is stored.

Returns

EventData – onset annotations

2.5.22 freesound_one_shot_percussive_sounds

Freesound One-Shot Percussive Sounds Dataset Loader

Dataset Info

Introduction:

This dataset contains 10254 one-shot (single event) percussive sounds from freesound.org, a timbral analysis computed by two different extractors (FreesoundExtractor from Essentia and AudioCommons Extractor), and a list of tags. There is also metadata information about the audio file, since the audio specifications are not the same along all the dataset tracks. The analysis data was used to train the generative model for "Neural Percussive Synthesis Parameterised by High-Level Timbral Features".

Dataset Construction:

To collect this dataset, the following steps were performed: * Freesound was queried with words associated with percussive instruments, such as "percussion", "kick", "wood" or "clave". Only sounds with less than one second of effective duration were selected. * This stage retrieved some audio clips that contained multiple sound events or that were of low quality. Therefore, we listened to all the retrieved sounds and manually discarded the sounds presenting one of these characteristics. For this, the percussive-annotator was used (https://github.com/xavierfav/percussive-annotator). This tool allows the user to annotate a dataset that focuses on percussive sounds. * The sounds were then cut or padded to have 1-second length, normalized and downsampled to 16kHz. * Finally, the sounds were analyzed with the AudioCommons Extractor, to obtain the AudioCommons timbral descriptors.

Authors and Contact:

This dataset was developed by António Ramires, Pritish Chadna, Xavier Favory, Emilia Gómez and Xavier Serra. Any questions related to this dataset please contact: António Ramires (antonio.ramires@upf.edu / aframires@gmail.com)

Acknowledgements:

This work has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 765068 (MIP-Frontiers). This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 770376 (TROMPA).

`class mirdata.datasets.freesound_one_shot_percussive_sounds.Dataset(`*data_home=None*, *version='default'*`)`

The Freesound One-Shot Percussive Sounds dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list

- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_audio`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.freesound_one_shot_percussive_sounds.load_audio

`load_file_metadata`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.freesound_one_shot_percussive_sounds.load_file_metadata

`load_multitracks`()

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

`load_tracks`()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
`NotImplementedError` – If the dataset does not support Tracks

mtrack_ids

Return track ids

Returns
list – A list of track ids

track_ids

Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.freesound_one_shot_percussive_sounds.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

Freesound one-shot percussive sounds track class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored. If *None*, looks for the data in the default directory, *~/mir_datasets/freesound_one_shot_percussive_sounds*

Variables

- **file_metadata_path** (*str*) – local path where the analysis file is stored and from where we get the file metadata
- **audio_path** (*str*) – local path where audio file is stored
- **track_id** (*str*) – track id
- **filename** (*str*) – filename of the track
- **username** (*str*) – username of the Freesound uploader of the track
- **license** (*str*) – link to license of the track file
- **tags** (*list*) – list of tags of the track
- **freesound_preview_urls** (*dict*) – dict of Freesound previews urls of the track
- **freesound_analysis** (*str*) – dict of analysis parameters computed in Freesound using Essentia extractor
- **audiocommons_analysis** (*str*) – dict of analysis parameters computed using AudioCommons Extractor

Other Parameters
file_metadata (*dict*) – metadata parameters of the track file in form of Python dictionary

property audio: Tuple[numpy.ndarray, float] | None

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.freesound_one_shot_percussive_sounds.load_audio(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load the track audio file.

Parameters
fhandle (*str*) – path to an audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.freesound_one_shot_percussive_sounds.load_file_metadata(*fhandle: TextIO*) → dict | None

Extract file metadata from analysis json file

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to f0 annotation file

Returns
analysis – track analysis dict

2.5.23 giantsteps_key

giantsteps_key Dataset Loader

Dataset Info

The GiantSteps+ EDM Key Dataset includes 600 two-minute sound excerpts from various EDM subgenres, annotated with single-key labels, comments and confidence levels by Daniel G. Camhi, and thoroughly revised and expanded by Ángel Faraldo at MTG UPF. Additionally, 500 tracks have been thoroughly analysed, containing pitch-class set descriptions, key changes, and additional modal changes. This dataset is a revision of the original GiantSteps Key Dataset, available in Github (<https://github.com/GiantSteps/giantsteps-key-dataset>) and initially described in:

```
Knees, P., Faraldo, Á., Herrera, P., Vogl, R., Böck, S., Hörschläger, F., Le Goff, M.␣
↪(2015).
Two Datasets for Tempo Estimation and Key Detection in Electronic Dance Music Annotated␣
↪from User Corrections.
In Proceedings of the 16th International Society for Music Information Retrieval␣
↪Conference, 364-370. Málaga, Spain.
```

The original audio samples belong to online audio snippets from Beatport, an online music store for DJ's and Electronic Dance Music Producers (<http:www.beatport.com>). If this dataset were used in further research, we would appreciate the citation of the current DOI (10.5281/zenodo.1101082) and the following doctoral dissertation, where a detailed description of the properties of this dataset can be found:

```
Ángel Faraldo (2017). Tonality Estimation in Electronic Dance Music: A Computational and␣
↪Musically Informed Examination.
PhD Thesis. Universitat Pompeu Fabra, Barcelona.
```

This dataset is mainly intended to assess the performance of computational key estimation algorithms in electronic dance music subgenres.

All the data of this dataset is licensed with Creative Commons Attribution Share Alike 4.0 International.

---

class mirdata.datasets.giantsteps_key.**Dataset**(*data_home=None*, *version='default'*)

The giantsteps_key dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

get_random_track_splits(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list

- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_track_splits()`

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license()`

Print the license

`load_artist(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.giantsteps_key.load_artist

`load_audio(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.giantsteps_key.load_audio

`load_genre(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.giantsteps_key.load_genre

`load_key(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.giantsteps_key.load_key

`load_multitracks()`

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

`load_tempo(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.giantsteps_key.load_tempo

`load_tracks()`

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
`NotImplementedError` – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns
list – A list of track ids

track_ids
Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)
Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.giantsteps_key.Track(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)
giantsteps_key track class

Parameters
track_id (*str*) – track id of the track

Variables

- **audio_path** (*str*) – track audio path
- **keys_path** (*str*) – key annotation path
- **metadata_path** (*str*) – sections annotation path
- **title** (*str*) – title of the track
- **track_id** (*str*) – track id

Other Parameters

- **key** (*str*) – musical key annotation
- **artists** (*list*) – list of artists involved
- **genres** (*dict*) – genres and subgenres
- **tempo** (*int*) – crowdsourced tempo annotations in beats per minute

property audio: Tuple[numpy.ndarray, float]
The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)
Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.giantsteps_key.**load_artist**(*fhandle: TextIO*) → List[str]

Load giantsteps_key tempo data from a file

Parameters
fhandle (*str or file-like*) – File-like object or path pointing to metadata annotation file

Returns
list – list of artists involved in the track.

mirdata.datasets.giantsteps_key.**load_audio**(*fpath: str*) → Tuple[numpy.ndarray, float]

Load a giantsteps_key audio file.

Parameters
fpath (*str*) – str pointing to an audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.giantsteps_key.**load_genre**(*fhandle: TextIO*) → Dict[str, List[str]]

Load giantsteps_key genre data from a file

Parameters
fhandle (*str or file-like*) – File-like object or path pointing to metadata annotation file

Returns
dict – {'genres': [...], 'subgenres': [...]}

mirdata.datasets.giantsteps_key.**load_key**(*fhandle: TextIO*) → str

Load giantsteps_key format key data from a file

Parameters
fhandle (*str or file-like*) – File like object or string pointing to key annotation file

Returns
str – loaded key data

mirdata.datasets.giantsteps_key.**load_tempo**(*fhandle: TextIO*) → str

Load giantsteps_key tempo data from a file

Parameters
fhandle (*str or file-like*) – File-like object or string pointing to metadata annotation file

Returns
str – loaded tempo data

2.5.24 giantsteps_tempo

giantsteps_tempo Dataset Loader

Dataset Info

GiantSteps tempo + genre is a collection of annotations for 664 2min(1) audio previews from www.beatport.com, created by Richard Vogl <richard.vogl@tuwien.ac.at> and Peter Knees <peter.knees@tuwien.ac.at>

references:

The audio files (664 files, size ~1gb) can be downloaded from http://www.beatport.com/ using the bash script:

https://github.com/GiantSteps/giantsteps-tempo-dataset/blob/master/audio_dl.sh

To download the files manually use links of the following form: http://geo-samples.beatport.com/lofi/<name of mp3 file> e.g.: http://geo-samples.beatport.com/lofi/5377710.LOFI.mp3

To convert the audio files to .wav use the script found at https://github.com/GiantSteps/giantsteps-tempo-dataset/blob/master/convert_audio.sh and run:

```
./convert_audio.sh
```

To retrieve the genre information, the JSON contained within the website was parsed. The tempo annotation was extracted from forum entries of people correcting the bpm values (i.e. manual annotation of tempo). For more information please refer to the publication [giantsteps_tempo_cit_1].

[giantsteps_tempo_cit_2] found some files without tempo. There are:

```
3041381.LOFI.mp3
3041383.LOFI.mp3
1327052.LOFI.mp3
```

Their v2 tempo is denoted as 0.0 in tempo and mirex and has no annotation in the JAMS format.

Most of the audio files are 120 seconds long. Exceptions are:

```
name              length (sec)
906760.LOFI.mp3   62
1327052.LOFI.mp3  70
4416506.LOFI.mp3  80
1855660.LOFI.mp3  119
3419452.LOFI.mp3  119
3577631.LOFI.mp3  119
```

---

class mirdata.datasets.giantsteps_tempo.**Dataset**(*data_home=None*, *version='default'*)

The giantsteps_tempo dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –

- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

get_random_track_splits(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

get_track_splits()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- **AttributeError** – If this dataset does not have tracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of track_ids

license()

Print the license

load_audio(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.giantsteps_tempo.load_audio

load_genre(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.giantsteps_tempo.load_genre

load_multitracks()

Load all multitracks in the dataset

Returns

dict – {*mtrack_id*: multitrack data}

Raises

NotImplementedError – If the dataset does not support Multitracks

load_tempo(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.giantsteps_tempo.load_tempo

load_tracks()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

mtrack_ids

Return track ids

Returns
list – A list of track ids

track_ids

Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.giantsteps_tempo.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

giantsteps_tempo track class

Parameters
track_id (*str*) – track id of the track

Variables

- **audio_path** (*str*) – track audio path
- **title** (*str*) – title of the track
- **track_id** (*str*) – track id
- **annotation_v1_path** (*str*) – track annotation v1 path
- **annotation_v2_path** (*str*) – track annotation v2 path

Other Parameters

- **genre** (*dict*) – Human-labeled metadata annotation
- **tempo** (*list*) – List of annotations.TempoData, ordered by confidence
- **tempo_v2** (*list*) – List of annotations.TempoData for version 2, ordered by confidence

property audio: Tuple[numpy.ndarray, float]

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

to_jams_v2()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.giantsteps_tempo.**load_audio**(*fhandle: str*) → Tuple[numpy.ndarray, float]

Load a giantsteps_tempo audio file.

Parameters
fhandle (*str or file-like*) – path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.giantsteps_tempo.**load_genre**(*fhandle: TextIO*) → str

Load genre data from a file

Parameters
path (*str*) – path to metadata annotation file

Returns
str – loaded genre data

mirdata.datasets.giantsteps_tempo.**load_tempo**(*fhandle: TextIO*) → *TempoData*

Load giantsteps_tempo tempo data from a file ordered by confidence

Parameters
fhandle (*str or file-like*) – File-like object or path to tempo annotation file

Returns
annotations.TempoData – Tempo data

2.5.25 good_sounds

Good-Sounds Dataset Loader

Dataset Info

The Good-Sounds dataset is born of the collaboration between the Music Technology Group and Korg. Good-Sounds [2, 16] is carried out recording a training dataset of single note excerpts including six classes of sounds per studied instrument. Twelve different instruments are recorded, as is shown in Table 2. For each instrument, the complete range of playable semitones is captured several times with various tonal characteristics. There are two classes: Good and Bad sounds. Bad sounds are divided into five sub-classes, one for each musical dimension stated by the expert musicians. Bad sounds are composed by examples of note recordings that are intentionally badly played. The last class includes examples of note recordings that are considered to be well played.

This dataset was created in the context of the Pablo project, partially funded by KORG Inc. It contains monophonic recordings of two kind of exercises: single notes and scales. The recordings were made in the Universitat Pompeu Fabra / Phonos recording studio by 15 different professional musicians, all of them holding a music degree and having some expertise in teaching. 12 different instruments were recorded using one or up to 4 different microphones (depending on the recording session). For all the instruments the whole set of playable semitones in the instrument is recorded several times with different tonal characteristics. Each note is recorded into a separate mono .flac audio file of 48kHz and 32 bits. The tonal characteristics are explained both in the the following section and the related publication. The database is meant for organizing the sounds in a handy way. It is organised in four different entities: sounds, takes, packs and ratings.

class mirdata.datasets.good_sounds.**Dataset**(*data_home=None*, *version='default'*)

The GOOD-SOUNDS dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

`cite()`

Print the reference

`property default_path`

Get the default path for the dataset

Returns
str – Local path to the dataset

`download(partial_download=None, force_overwrite=False, cleanup=False, allow_invalid_checksum=False)`

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- `ValueError` – if invalid keys are passed to partial_download
- `IOError` – if a downloaded file's checksum is different from expected

`get_mtrack_splits()`

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- `AttributeError` – If this dataset does not have multitracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits(splits, seed=42, split_names=None)`

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_random_track_splits(splits, seed=42, split_names=None)`

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_track_splits()`

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license()`

Print the license

`load_audio(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.good_sounds.load_audio

`load_multitracks()`

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

`load_tracks()`

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
`NotImplementedError` – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns
list – A list of track ids

`track_ids`

Return track ids

Returns
list – A list of track ids

`validate(verbose=True)`

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

`class mirdata.datasets.good_sounds.Track(track_id, data_home, dataset_name, index, metadata)`

GOOD-SOUNDS Track class

Parameters
track_id (*str*) – track id of the track

Variables
audio_path (*str*) – Path to the audio file

Other Parameters

- **ratings_info** (*dict*) – A dictionary containing the entity Ratings.
- **Some musicians self-rated their performance in a 0-10 goodness scale for the user evaluation of the first project**
- **prototype. Please read the paper for more detailed information.** –
  - id
  - mark: the rate or score.
  - type: the klass of the sound. Related to the tags of the sound.
  - created_at
  - comments
  - sound_id
  - rater: the musician who rated the sound.
- **pack_info** (*dict*) – A dictionary containing the entity Pack. A pack is a group of sounds from the same recording session. The audio files are organised in the sound_files directory in subfolders with the pack name to which they belong. The following metadata is associated with the entity Pack. - id - name - description
- **sound_info** (*dict*) – A dictionary containing the entity Sound. A sound can have several takes as some of them were recorded using different microphones at the same time. The following metadata is associated with the entity Sound. - id - instrument: flute, cello, clarinet, trumpet, violin, sax_alto, sax_tenor, sax_baritone, sax_soprano, oboe, piccolo, bass - note - octave - dynamics: for some sounds, the musical notation of the loudness level (p, mf, f..) - recorded_at: recording date and time - location: recording place - player: the musician who recorded. For detailed information about the musicians contact us. - bow_velocity: for some string instruments the velocity of the bow (slow, medieum, fast) - bridge_position: for some string instruments the position of the bow (tasto, middle, ponticello) - string: for some string instruments the number of the string in which the sound it's played (1: lowest in pitch) - csv_file: used for creation of the DB - csv_id: used for creation of the DB - pack_filename: used for creation of the DB - pack_id: used for creation of the DB - attack: for single notes, manual annotation of the onset in samples. - decay: for single notes, manual annotation of the decay in samples. - sustain: for single notes, manual annotation of the beginnig of the sustained part in samples. - release: for single notes, manual annotation of the beginnig of the release part in samples. - offset: for single notes, manual annotation of the offset in samples - reference: 1 if sound was used to create the models in the good-sounds project, 0

if not. - Other tags regarding tonal characteristics are also available. - comments: if any - semitone: midi note - pitch_reference: the reference pitch - klass: user generated tags of the tonal qualities of the sound. They also contain information about the exercise, that could be single note or scale. * "good-sound": good examples of single note * "bad": bad example of one of the sound attributes defined in the project (please read the papers for a detailed explanation) * "scale-good": good example of a one octave scale going up and down (15 notes). If the scale is minor a tagged "minor" is also available. * "scale-bad": bad example scale of one of the sounds defined in the project. (15 notes up and down).

- **take_info** (*dict*) – A dictionary containing the entity Take. A sound can have several takes as some of them were recorded using different microphones at the same time. Each take has an associated audio file. The annotations. Each take has an associated audio file. The following metadata is associated with the entity Sound. - id - microphone - filename: the name of the associated audio file - original_filename: - freesound_id: for some sounds uploaded to freesound.org - sound_id: the id of the sound in the DB - goodsound_id: for some of the sounds available in good-sounds.org
- **microphone** (*str*) – the microphone used to record the take.
- **instrument** (*str*) – the instrument recorded (flute, cello, clarinet, trumpet, violin, sax_alto, sax_tenor, sax_baritone, sax_soprano, oboe, piccolo, bass).
- **klass** (*str*) – user generated tags of the tonal qualities of the sound. They also contain information about the exercise, that could be single note or scale. * "good-sound": good examples of single note * "bad": bad example of one of the sound attributes defined in the project (please read the papers for a detailed explanation) * "scale-good": good example of a one octave scale going up and down (15 notes). If the scale is minor a tagged "minor" is also available. * "scale-bad": bad example scale of one of the sounds defined in the project. (15 notes up and down).
- **semitone** (*int*) – midi note
- **pitch_reference** (*int*) – the reference pitch

property audio: Tuple[numpy.ndarray, float] | None

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

mirdata.datasets.good_sounds.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a GOOD-SOUNDS audio file.

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to an audio file

Returns

- np.ndarray - the mono audio signal

- float - The sample rate of the audio file

2.5.26 groove_midi

Groove MIDI Loader

Dataset Info

The Groove MIDI Dataset (GMD) is composed of 13.6 hours of aligned MIDI and synthesized audio of human-performed, tempo-aligned expressive drumming. The dataset contains 1,150 MIDI files and over 22,000 measures of drumming.

To enable a wide range of experiments and encourage comparisons between methods on the same data, Gillick et al. created a new dataset of drum performances recorded in MIDI format. They hired professional drummers and asked them to perform in multiple styles to a click track on a Roland TD-11 electronic drum kit. They also recorded the aligned, high-quality synthesized audio from the TD-11 and include it in the release.

The Groove MIDI Dataset (GMD), has several attributes that distinguish it from existing ones:

- The dataset contains about 13.6 hours, 1,150 MIDI files, and over 22,000 measures of drumming.
- Each performance was played along with a metronome set at a specific tempo by the drummer.
- The data includes performances by a total of 10 drummers, with more than 80% of duration coming from hired professionals. The professionals were able to improvise in a wide range of styles, resulting in a diverse dataset.
- The drummers were instructed to play a mix of long sequences (several minutes of continuous playing) and short beats and fills.
- Each performance is annotated with a genre (provided by the drummer), tempo, and anonymized drummer ID.
- Most of the performances are in 4/4 time, with a few examples from other time signatures.
- Four drummers were asked to record the same set of 10 beats in their own style. These are included in the test set split, labeled eval-session/groove1-10.
- In addition to the MIDI recordings that are the primary source of data for the experiments in this work, the authors captured the synthesized audio outputs of the drum set and aligned them to within 2ms of the corresponding MIDI files.

A train/validation/test split configuration is provided for easier comparison of model accuracy on various tasks.

The dataset is made available by Google LLC under a Creative Commons Attribution 4.0 International (CC BY 4.0) License.

For more details, please visit: http://magenta.tensorflow.org/datasets/groove

`class mirdata.datasets.groove_midi.Dataset(`*data_home=None*, *version='default'*`)`

The groove_midi dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –

- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

get_random_track_splits(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

get_track_splits()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- **AttributeError** – If this dataset does not have tracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of track_ids

license()

Print the license

load_audio(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.groove_midi.load_audio

load_beats(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.groove_midi.load_beats

load_drum_events(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.groove_midi.load_drum_events

load_midi(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.groove_midi.load_midi

load_multitracks()

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
NotImplementedError – If the dataset does not support Multitracks

`load_tracks()`

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns
list – A list of track ids

`track_ids`

Return track ids

Returns
list – A list of track ids

`validate(verbose=True)`

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

`class mirdata.datasets.groove_midi.Track(track_id, data_home, dataset_name, index, metadata)`

Groove MIDI Track class

Parameters
track_id (*str*) – track id of the track

Variables

- **drummer** (*str*) – Drummer id of the track (ex. 'drummer1')
- **session** (*str*) – Type of session (ex. 'session1', 'eval_session')
- **track_id** (*str*) – track id of the track (ex. 'drummer1/eval_session/1')
- **style** (*str*) – Style (genre, groove type) of the track (ex. 'funk/groove1')
- **tempo** (*int*) – track tempo in beats per minute (ex. 138)
- **beat_type** (*str*) – Whether the track is a beat or a fill (ex. 'beat')
- **time_signature** (*str*) – Time signature of the track (ex. '4-4', '6-8')
- **midi_path** (*str*) – Path to the midi file
- **audio_path** (*str*) – Path to the audio file
- **duration** (*float*) – Duration of the midi file in seconds

- **split** (*str*) – Whether the track is for a train/valid/test set. One of 'train', 'valid' or 'test'.

Other Parameters

- **beats** (*BeatData*) – Machine-generated beat annotations
- **drum_events** (*EventData*) – Annotated drum kit events
- **midi** (*pretty_midi.PrettyMIDI*) – object containing MIDI information

property audio: Tuple[numpy.ndarray | None, float | None]

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.groove_midi.**load_audio**(*path: str*) → Tuple[numpy.ndarray | None, float | None]

Load a Groove MIDI audio file.

Parameters
path – path to an audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.groove_midi.**load_beats**(*midi_path*, *midi=None*)

Load beat data from the midi file.

Parameters

- **midi_path** (*str*) – path to midi file
- **midi** (*pretty_midi.PrettyMIDI*) – pre-loaded midi object or None if None, the midi object is loaded using midi_path

Returns
annotations.BeatData – machine generated beat data

mirdata.datasets.groove_midi.**load_drum_events**(*midi_path*, *midi=None*)

Load drum events from the midi file.

Parameters

- **midi_path** (*str*) – path to midi file

- **midi** (*pretty_midi.PrettyMIDI*) – pre-loaded midi object or None if None, the midi object is loaded using midi_path

Returns
annotations.EventData – drum event data

`mirdata.datasets.groove_midi.load_midi(fhandle: BinaryIO) → pretty_midi.PrettyMIDI | None`

Load a Groove MIDI midi file.

Parameters
fhandle (*str or file-like*) – File-like object or path to midi file

Returns
midi_data (pretty_midi.PrettyMIDI) – pretty_midi object

2.5.27 gtzan_genre

GTZAN-Genre Dataset Loader

Dataset Info

This dataset was used for the well known genre classification paper:

```
"Musical genre classification of audio signals " by G. Tzanetakis and
P. Cook in IEEE Transactions on Audio and Speech Processing 2002.
```

The dataset consists of 1000 audio tracks each 30 seconds long. It contains 10 genres, each represented by 100 tracks. The tracks are all 22050 Hz mono 16-bit audio files in .wav format.

`class mirdata.datasets.gtzan_genre.Dataset(data_home=None, version='default')`

The gtzan_genre dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

`choice_multitrack()`

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_audio`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.gtzan_genre.load_audio

`load_multitracks`()

Load all multitracks in the dataset

Returns

dict – {*mtrack_id*: multitrack data}

Raises

`NotImplementedError` – If the dataset does not support Multitracks

`load_tracks`()

Load all tracks in the dataset

Returns

dict – {*track_id*: track data}

Raises

`NotImplementedError` – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns

list – A list of track ids

`track_ids`

Return track ids

Returns

list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.gtzan_genre.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

gtzan_genre Track class

Parameters
track_id (*str*) – track id of the track

Variables

- **audio_path** (*str*) – path to the audio file
- **genre** (*str*) – annotated genre
- **track_id** (*str*) – track id

Other Parameters

- **beats** (*BeatData*) – human-labeled beat annotations
- **tempo** (*float*) – global tempo annotations

property audio: Tuple[numpy.ndarray, float] | None

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.gtzan_genre.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a GTZAN audio file.

Parameters
fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal

- float - The sample rate of the audio file

mirdata.datasets.gtzan_genre.**load_beats**(*fhandle: TextIO*) → *BeatData*

Load GTZAN format beat data from a file

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to a beat annotation file

Returns
BeatData – loaded beat data

mirdata.datasets.gtzan_genre.**load_tempo**(*fhandle: TextIO*) → float

Load GTZAN format tempo data from a file

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to a beat annotation file

Returns
tempo (float) – loaded tempo data

2.5.28 guitarset

GuitarSet Loader

Dataset Info

GuitarSet provides audio recordings of a variety of musical excerpts played on an acoustic guitar, along with time-aligned annotations including pitch contours, string and fret positions, chords, beats, downbeats, and keys.

GuitarSet contains 360 excerpts that are close to 30 seconds in length. The 360 excerpts are the result of the following combinations:

- 6 players
- 2 versions: comping (harmonic accompaniment) and soloing (melodic improvisation)
- 5 styles: Rock, Singer-Songwriter, Bossa Nova, Jazz, and Funk
- 3 Progressions: 12 Bar Blues, Autumn Leaves, and Pachelbel Canon.
- 2 Tempi: slow and fast.

The tonality (key) of each excerpt is sampled uniformly at random.

GuitarSet was recorded with the help of a hexaphonic pickup, which outputs signals for each string separately, allowing automated note-level annotation. Excerpts are recorded with both the hexaphonic pickup and a Neumann U-87 condenser microphone as reference. 3 audio recordings are provided with each excerpt with the following suffix:

- hex: original 6 channel wave file from hexaphonic pickup
- hex_cln: hex wave files with interference removal applied
- mic: monophonic recording from reference microphone
- mix: monophonic mixture of original 6 channel file

Each of the 360 excerpts has an accompanying JAMS file which stores 16 annotations. Pitch:

- 6 pitch_contour annotations (1 per string)
- 6 midi_note annotations (1 per string)

Beat and Tempo:

- 1 beat_position annotation
- 1 tempo annotation

Chords:

- 2 chord annotations: instructed and performed. The instructed chord annotation is a digital version of the lead sheet that's provided to the player, and the performed chord annotations are inferred from note annotations, using segmentation and root from the digital lead sheet annotation.

For more details, please visit: http://github.com/marl/guitarset/

---

class mirdata.datasets.guitarset.**Dataset**(*data_home=None*, *version='default'*)

The guitarset dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns

str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.

- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

`get_mtrack_splits()`

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits(splits, seed=42, split_names=None)`

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_random_track_splits(splits, seed=42, split_names=None)`

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_track_splits()`

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- **AttributeError** – If this dataset does not have tracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license()`

Print the license

`load_audio(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.guitarset.load_audio

`load_beats(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.guitarset.load_beats

`load_chords(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.guitarset.load_chords

`load_key_mode(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.guitarset.load_key_mode

`load_multitrack_audio(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.guitarset.load_multitrack_audio

`load_multitracks()`

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

`load_notes(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.guitarset.load_notes

`load_pitch_contour(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.guitarset.load_pitch_contour

`load_tracks()`

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
`NotImplementedError` – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns
list – A list of track ids

`track_ids`

Return track ids

Returns
list – A list of track ids

`validate(verbose=True)`

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.guitarset.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

guitarset Track class

Parameters
track_id (*str*) – track id of the track

Variables

- **audio_hex_cln_path** (*str*) – path to the debleeded hex wave file
- **audio_hex_path** (*str*) – path to the original hex wave file
- **audio_mic_path** (*str*) – path to the mono wave via microphone
- **audio_mix_path** (*str*) – path to the mono wave via downmixing hex pickup
- **jams_path** (*str*) – path to the jams file
- **mode** (*str*) – one of ['solo', 'comp'] For each excerpt, players are asked to first play in 'comp' mode and later play a 'solo' version on top of the already recorded comp.
- **player_id** (*str*) – ID of the different players. one of ['00', '01', … , '05']
- **style** (*str*) – one of ['Jazz', 'Bossa Nova', 'Rock', 'Singer-Songwriter', 'Funk']
- **tempo** (*float*) – BPM of the track
- **track_id** (*str*) – track id

Other Parameters

- **beats** (*BeatData*) – beat positions
- **leadsheet_chords** (*ChordData*) – chords as written in the leadsheet
- **inferred_chords** (*ChordData*) – chords inferred from played transcription
- **key_mode** (*KeyData*) – key and mode
- **pitch_contours** (*dict*) – Pitch contours per string - 'E': F0Data(…) - 'A': F0Data(…) - 'D': F0Data(…) - 'G': F0Data(…) - 'B': F0Data(…) - 'e': F0Data(…)
- **multif0** (*MultiF0Data*) – all pitch contour data as one multif0 annotation
- **notes** (*dict*) – Notes per string - 'E': NoteData(…) - 'A': NoteData(…) - 'D': NoteData(…) - 'G': NoteData(…) - 'B': NoteData(…) - 'e': NoteData(…)
- **notes_all** (*NoteData*) – all note data as one note annotation

property **audio_hex**: Tuple[numpy.ndarray, float] | None

Hexaphonic audio (6-channels) with one channel per string

Returns

- np.ndarray - audio signal
- float - sample rate

property audio_hex_cln: Tuple[numpy.ndarray, float] | None

Hexaphonic audio (6-channels) with one channel per string
after bleed removal

Returns

- np.ndarray - audio signal
- float - sample rate

property audio_mic: Tuple[numpy.ndarray, float] | None

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

property audio_mix: Tuple[numpy.ndarray, float] | None

Mixture audio (mono)

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.guitarset.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a Guitarset audio file.

Parameters
fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.guitarset.**load_beats**(*fhandle: TextIO*) → *BeatData*

Load a Guitarset beats annotation.

Parameters
fhandle (*str or file-like*) – File-like object or path of the jams annotation file

Returns
BeatData – Beat data

mirdata.datasets.guitarset.**load_chords**(*jams_path*, *leadsheet_version*)

Load a guitarset chord annotation.

Parameters

- **jams_path** (*str*) – path to the jams annotation file
- **leadsheet_version** (*Bool*) – Whether or not to load the leadsheet version of the chord annotation If False, load the infered version.

Returns

ChordData – Chord data

mirdata.datasets.guitarset.**load_key_mode**(*fhandle: TextIO*) → *KeyData*

Load a Guitarset key-mode annotation.

Parameters

fhandle (*str or file-like*) – File-like object or path of the jams annotation file

Returns

KeyData – Key data

mirdata.datasets.guitarset.**load_multitrack_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a Guitarset multitrack audio file.

Parameters

fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.guitarset.**load_notes**(*jams_path*, *string_num*)

Load a guitarset note annotation for a given string

Parameters

- **jams_path** (*str*) – path to the jams annotation file
- **string_num** (*int), in range(6*) – Which string to load. 0 is the Low E string, 5 is the high e string.

Returns

NoteData – Note data for the given string

mirdata.datasets.guitarset.**load_pitch_contour**(*jams_path*, *string_num*)

Load a guitarset pitch contour annotation for a given string

Parameters

- **jams_path** (*str*) – path to the jams annotation file
- **string_num** (*int), in range(6*) – Which string to load. 0 is the Low E string, 5 is the high e string.

Returns

F0Data – Pitch contour data for the given string

2.5.29 haydn_op20

haydn op20 Dataset Loader

Dataset Info

This dataset accompanies the Master Thesis from Nestor Napoles. It is a manually-annotated corpus of harmonic analysis in harm syntax.

This dataset contains 30 pieces composed by Joseph Haydn in symbolic format, which have each been manually annotated with harmonic analyses.

`class mirdata.datasets.haydn_op20.Dataset(data_home=None, version='default')`

The haydn op20 dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

`choice_multitrack()`

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

`choice_track()`

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

`cite()`

Print the reference

`property default_path`

Get the default path for the dataset

Returns

str – Local path to the dataset

`download(partial_download=None, force_overwrite=False, cleanup=False, allow_invalid_checksum=False)`

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded

- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

get_random_track_splits(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

get_track_splits()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- **AttributeError** – If this dataset does not have tracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license()`

Print the license

`load_chords(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.haydn_op20.load_chords

`load_chords_music21(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.haydn_op20.load_chords_music21

`load_key(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.haydn_op20.load_key

`load_key_music21(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.haydn_op20.load_key_music21

`load_midi_path(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.haydn_op20.convert_and_save_to_midi

`load_multitracks()`

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

`load_roman_numerals(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.haydn_op20.load_roman_numerals

`load_score(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.haydn_op20.load_score

`load_tracks()`

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
`NotImplementedError` – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns
list – A list of track ids

`track_ids`

Return track ids

Returns
list – A list of track ids

`validate(verbose=True)`

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.haydn_op20.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

haydn op20 track class

Parameters
track_id (*str*) – track id of the track

Variables

- **title** (*str*) – title of the track
- **track_id** (*str*) – track id
- **humdrum_annotated_path** (*str*) – path to humdrum annotated score

Other Parameters

- **keys** (*KeyData*) – annotated local keys.
- **keys_music21** (*list*) – annotated local keys.
- **roman_numerals** (*list*) – annotated roman_numerals.
- **chords** (*ChordData*) – annotated chords.
- **chords_music21** (*list*) – annotated chords.
- **duration** (*int*) – relative duration
- **midi_path** (*str*) – path to midi
- **score** (*music21.stream.Score*) – music21 score

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.haydn_op20.**convert_and_save_to_midi**(*fpath: TextIO*)

convert to midi file and return the midi path

Parameters
fpath (*str or file-like*) – path to score file

Returns
str – midi file path

Deprecated since version 0.3.4: convert_and_save_to_midi is deprecated and will be removed in a future version

mirdata.datasets.haydn_op20.**load_chords**(*fhandle: TextIO*, *resolution: int = 28*)

Load haydn op20 chords data from a file

Parameters

- **fhandle** (*str or file-like*) – path to chord annotations
- **resolution** (*int*) – the number of pulses, or ticks, per quarter note (PPQ)

Returns

ChordData – chord annotations

mirdata.datasets.haydn_op20.**load_chords_music21**(*fhandle: TextIO*, *resolution: int = 28*)

Load haydn op20 chords data from a file in music21 format

Parameters

- **fhandle** (*str or file-like*) – path to chord annotations
- **resolution** (*int*) – the number of pulses, or ticks, per quarter note (PPQ)

Returns

list – musical chords data and relative time (offset (Music21Object.offset) * resolution) [(time in PPQ, chord)]

mirdata.datasets.haydn_op20.**load_key**(*fhandle: TextIO*, *resolution=28*)

Load haydn op20 key data from a file

Parameters

- **fhandle** (*str or file-like*) – path to key annotations
- **resolution** (*int*) – the number of pulses, or ticks, per quarter note (PPQ)

Returns

KeyData – loaded key data

mirdata.datasets.haydn_op20.**load_key_music21**(*fhandle: TextIO*, *resolution=28*)

Load haydn op20 key data from a file in music21 format

Parameters

- **fhandle** (*str or file-like*) – path to key annotations
- **resolution** (*int*) – the number of pulses, or ticks, per quarter note (PPQ)

Returns

list – musical key data and relative time (offset (Music21Object.offset) * resolution) [(time in PPQ, local key)]

mirdata.datasets.haydn_op20.**load_roman_numerals**(*fhandle: TextIO*, *resolution=28*)

Load haydn op20 roman numerals data from a file

Parameters

- **fhandle** (*str or file-like*) – path to roman numeral annotations
- **resolution** (*int*) – the number of pulses, or ticks, per quarter note (PPQ)

Returns

list – musical roman numerals data and relative time (offset (Music21Object.offset) * resolution) [(time in PPQ, roman numerals)]

mirdata.datasets.haydn_op20.**load_score**(*fhandle: TextIO*)

Load haydn op20 score with annotations from a file with music21 format (music21.stream.Score).

Parameters
fhandle (*str or file-like*) – path to score

Returns
music21.stream.Score – score in music21 format

2.5.30 idmt_smt_audio_effects

IDMT-SMT-Audio-Effects Dataset Loader

Dataset Info

IDMT-SMT-Audio-Effects is a large database for automatic detection of audio effects in recordings of electric guitar and bass and related signal processing. The overall duration of the audio material is approx. 30 hours.

The dataset consists of 55044 WAV files (44.1 kHz, 16bit, mono) with single recorded notes:

20592 monophonic bass notes 20592 monophonic guitar notes 13860 polyphonic guitar sounds Overall, 11 different audio effects are incorporated: feedback delay, slapback delay, reverb, chorus, flanger, phaser, tremolo, vibrato, distortion, overdrive, no effect (unprocessed notes/sounds)

2 different electric guitars and 2 different electric bass guitars, each with two different pick-up settings and up to three different plucking styles (finger plucked - hard, finger plucked - soft, picked) were used for recording. The notes cover the common pitch range of a 4-string bass guitar from E1 (41.2 Hz) to G3 (196.0 Hz) or the common pitch range of a 6-string electric guitar from E2 (82.4 Hz) to E5 (659.3 Hz). Effects processing was performed using a digital audio workstation and a variety of mostly freely available effect plugins.

To organize the database, lists in XML format are used, which record all relevant information and are provided with the database as well as a summary of the used effect plugins and parameter settings.

In addition, most of this information is also encoded in the first part of the file name of the audio files using a simple alpha-numeric encoding scheme. The second part of the file name contains unique identification numbers. This provides an option for fast and flexible structuring of the data for various purposes.

DOI 10.5281/zenodo.7544032

---

class mirdata.datasets.idmt_smt_audio_effects.**Dataset**(*data_home=None*, *version='default'*)

The IDMT-SMT-Audio Effect dataset.

Parameters

- **data_home** (*str*) – Directory where the dataset is located or will be downloaded.
- **version** (*str*) – Dataset version. Default is "default".

Variables

- **name** (*str*) – Name of the dataset.
- **track_class** (*Type[core.Track]*) – Track type.
- **bibtex** (*str or None*) – BibTeX citation.
- **indexes** (*dict or None*) – Available versions.
- **remotes** (*dict or None*) – Data to be downloaded.
- **download_info** (*str*) – Instructions for downloading the dataset.

- **license_info** (*str*) – Dataset license.
- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** – the identifier of the dataset
- **bibtex** – dataset citation/s in bibtex format
- **indexes** –
- **remotes** – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download the dataset

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits`(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_multitracks`()

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

`load_tracks`()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

`mtrack_ids`
Return track ids

Returns
list – A list of track ids

`track_ids`
Return track ids

Returns
list – A list of track ids

`validate`(*verbose=True*)
Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

`class mirdata.datasets.idmt_smt_audio_effects.Track`(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)
IDMT-SMT-Audio-Effects track class.

Parameters

- **track_id** (*str*) – track id of the track.
- **data_home** (*str*) – Local path where the dataset is stored.
- **dataset_name** (*str*) – Name of the dataset.
- **index** (*Dict*) – Index dictionary.
- **metadata** (*Dict*) – Metadata dictionary.

Variables

- **audio_path** (*str*) – path to audio file.
- **instrument** (*str*) – instrument used to record the track.
- **midi_nr** (*int*) – midi number of the note.
- **fx_group** (*int*) – effect group number.
- **fx_type** (*int*) – effect type number.
- **fx_setting** (*int*) – effect setting number.

`property audio: Tuple[numpy.ndarray, float] | None`
The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.idmt_smt_audio_effects.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a IDMT-SMT-Audio Effect track.

Parameters
fhandle (*Union[str, BinaryIO]*) – Path to audio file or file-like object.

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

2.5.31 ikala

iKala Dataset Loader

Dataset Info

The iKala dataset is comprised of 252 30-second excerpts sampled from 206 iKala songs (plus 100 hidden excerpts reserved for MIREX). The music accompaniment and the singing voice are recorded at the left and right channels respectively and can be found under the Wavfile directory. In addition, the human-labeled pitch contours and timestamped lyrics can be found under PitchLabel and Lyrics respectively.

For more details, please visit: http://mac.citi.sinica.edu.tw/ikala/

---

class mirdata.datasets.ikala.**Dataset**(*data_home=None*, *version='default'*)

The ikala dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track

- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list

- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_f0`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.ikala.load_f0

`load_instrumental_audio`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.ikala.load_instrumental_audio

`load_lyrics`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.ikala.load_lyrics

`load_mix_audio`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.ikala.load_mix_audio

`load_multitracks`()

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

`load_notes`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.ikala.load_notes

load_pronunciations(*args, **kwargs)

Deprecated since version 0.3.4: Use mirdata.datasets.ikala.load_pronunciations

load_tracks()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

load_vocal_audio(*args, **kwargs)

Deprecated since version 0.3.4: Use mirdata.datasets.ikala.load_vocal_audio

mtrack_ids

Return track ids

Returns
list – A list of track ids

track_ids

Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.ikala.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

ikala Track class

Parameters
track_id (*str*) – track id of the track

Variables

- **audio_path** (*str*) – path to the track's audio file
- **f0_path** (*str*) – path to the track's f0 annotation file
- **notes_pyin_path** (*str*) – path to the note annotation file
- **lyrics_path** (*str*) – path to the track's lyric annotation file
- **section** (*str*) – section. Either 'verse' or 'chorus'
- **singer_id** (*str*) – singer id
- **song_id** (*str*) – song id of the track
- **track_id** (*str*) – track id

Other Parameters

- **f0** (*F0Data*) – human-annotated singing voice pitch
- **notes_pyin** (*NoteData*) – notes estimated by the pyin algorithm
- **lyrics** (*LyricsData*) – human-annotated lyrics
- **pronunciations** (*LyricsData*) – human-annotation lyric pronunciations

`get_path`(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

`property instrumental_audio: Tuple[numpy.ndarray, float] | None`

instrumental audio (mono)

Returns

- np.ndarray - audio signal
- float - sample rate

`property mix_audio: Tuple[numpy.ndarray, float] | None`

mixture audio (mono)

Returns

- np.ndarray - audio signal
- float - sample rate

`to_jams`()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

`property vocal_audio: Tuple[numpy.ndarray, float] | None`

solo vocal audio (mono)

Returns

- np.ndarray - audio signal
- float - sample rate

`mirdata.datasets.ikala.load_f0`(*fhandle: TextIO*) → *F0Data*

Load an ikala f0 annotation

Parameters
fhandle (*str or file-like*) – File-like object or path to f0 annotation file

Raises
IOError – If f0_path does not exist

Returns
F0Data – the f0 annotation data

mirdata.datasets.ikala.**load_instrumental_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load ikala instrumental audio

Parameters
fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - audio signal
- float - sample rate

mirdata.datasets.ikala.**load_lyrics**(*fhandle: TextIO*) → *LyricData*

Load an ikala lyrics annotation

Parameters
fhandle (*str or file-like*) – File-like object or path to lyric annotation file

Raises
IOError – if lyrics_path does not exist

Returns
LyricData – lyric annotation data

mirdata.datasets.ikala.**load_mix_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load an ikala mix.

Parameters
fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - audio signal
- float - sample rate

mirdata.datasets.ikala.**load_notes**(*fhandle: TextIO*) → *NoteData* | None

load a note annotation file

Parameters
fhandle (*str or file-like*) – str or file-like to note annotation file

Raises
IOError – if file doesn't exist

Returns
NoteData – note annotation

mirdata.datasets.ikala.**load_pronunciations**(*fhandle: TextIO*) → *LyricData*

Load an ikala pronunciation annotation

Parameters
fhandle (*str or file-like*) – File-like object or path to lyric annotation file

Raises
IOError – if lyrics_path does not exist

Returns
LyricData – pronunciation annotation data

mirdata.datasets.ikala.**load_vocal_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load ikala vocal audio

Parameters
fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - audio signal
- float - sample rate

2.5.32 irmas

IRMAS Loader

Dataset Info

IRMAS: a dataset for instrument recognition in musical audio signals

This dataset includes musical audio excerpts with annotations of the predominant instrument(s) present. It was used for the evaluation in the following article:

```
Bosch, J. J., Janer, J., Fuhrmann, F., & Herrera, P. "A Comparison of Sound Segregation␣
↪Techniques for
Predominant Instrument Recognition in Musical Audio Signals", in Proc. ISMIR (pp. 559-
↪564), 2012.
```

IRMAS is intended to be used for training and testing methods for the automatic recognition of predominant instruments in musical audio. The instruments considered are: cello, clarinet, flute, acoustic guitar, electric guitar, organ, piano, saxophone, trumpet, violin, and human singing voice. This dataset is derived from the one compiled by Ferdinand Fuhrmann in his PhD thesis, with the difference that we provide audio data in stereo format, the annotations in the testing dataset are limited to specific pitched instruments, and there is a different amount and lenght of excerpts from the original dataset.

The dataset is split into training and test data.

Training data

Total audio samples: 6705 They are excerpts of 3 seconds from more than 2000 distinct recordings.

Audio specifications

- Sampling frequency: 44.1 kHz
- Bit-depth: 16 bit
- Audio format: .wav

IRMAS Dataset trainig samples are annotated by storing the information of each track in their filenames.

- Predominant instrument:
  - The annotation of the predominant instrument of each excerpt is both in the name of the containing folder, and in the file name: cello (cel), clarinet (cla), flute (flu), acoustic guitar (gac), electric guitar (gel), organ (org), piano (pia), saxophone (sax), trumpet (tru), violin (vio), and human singing voice (voi).
  - The number of files per instrument are: cel(388), cla(505), flu(451), gac(637), gel(760), org(682), pia(721), sax(626), tru(577), vio(580), voi(778).
- Drum presence
  - Additionally, some of the files have annotation in the filename regarding the presence ([dru]) or non presence([nod]) of drums.

- The annotation of the musical genre:
  - country-folk ([cou_fol])
  - classical ([cla]),
  - pop-rock ([pop_roc])
  - latin-soul ([lat_sou])
  - jazz-blues ([jaz_blu]).

Testing data

Total audio samples: 2874

Audio specifications

- Sampling frequency: 44.1 kHz
- Bit-depth: 16 bit
- Audio format: .wav

IRMAS Dataset testing samples are annotated by the following basis:

- Predominant instrument:

  The annotations for an excerpt named: "excerptName.wav" are given in "excerptName.txt". More than one instrument may be annotated in each excerpt, one label per line. This part of the dataset contains excerpts from a diversity of western musical genres, with varied instrumentations, and it is derived from the original testing dataset from Fuhrmann (http://www.dtic.upf.edu/~ffuhrmann/PhD/). Instrument nomenclatures are the same as the training dataset.

Dataset compiled by Juan J. Bosch, Ferdinand Fuhrmann, Perfecto Herrera, Music Technology Group - Universitat Pompeu Fabra (Barcelona).

The IRMAS dataset is offered free of charge for non-commercial use only. You can not redistribute it nor modify it. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License

For more details, please visit: https://www.upf.edu/web/mtg/irmas

---

class mirdata.datasets.irmas.**Dataset**(*data_home=None*, *version='default'*)

The irmas dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list

- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_audio`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.irmas.load_audio

`load_multitracks`()

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

`load_pred_inst`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.irmas.load_pred_inst

`load_tracks`()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
`NotImplementedError` – If the dataset does not support Tracks

mtrack_ids

Return track ids

Returns
list – A list of track ids

track_ids

Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.irmas.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

IRMAS track class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored. If *None*, looks for the data in the default directory, *~/mir_datasets/Mridangam-Stroke*

Variables

- **track_id** (*str*) – track id
- **predominant_instrument** (*list*) – Training tracks predominant instrument
- **train** (*bool*) – flag to identify if the track is from the training of the testing dataset
- **genre** (*str*) – string containing the namecode of the genre of the track.
- **drum** (*bool*) – flag to identify if the track contains drums or not.
- **split** (*str*) – data split ("train" or "test")

Other Parameters
instrument (*list*) – list of predominant instruments as str

property audio: Tuple[numpy.ndarray, float] | None

The track's audio signal

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

`to_jams()`
the track's data in jams format

Returns
jams.JAMS – return track data in jam format

`mirdata.datasets.irmas.load_audio(fhandle: BinaryIO) → Tuple[numpy.ndarray, float]`
Load a IRMAS dataset audio file.

Parameters
fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

`mirdata.datasets.irmas.load_pred_inst(fhandle: TextIO) → List[str]`
Load predominant instrument of track

Parameters
fhandle (*str or file-like*) – File-like object or path where the test annotations are stored.

Returns
list(str) – test track predominant instrument(s) annotations

2.5.33 mtg_jamendo_autotagging_moodtheme

MTG jamendo autotagging moodtheme Dataset Loader

Dataset Info

The MTG Jamendo autotagging mood/theme Dataset is a new open dataset for music auto-tagging. It is built using music available at Jamendo under Creative Commons licenses and tags provided by content uploaders. The dataset contains 18,486 full audio tracks with 195 tags from mood/theme. It is provided five fixed data splits for a better and fair replication. For more information please visit: https://github.com/MTG/mtg-jamendo-dataset .

The moodtheme tags are:

action, adventure, advertising, ambiental, background, ballad, calm, children, christmas, commercial, cool, corporate, dark, deep, documentary, drama, dramatic, dream, emotional, energetic, epic, fast, film, fun, funny, game, groovy, happy, heavy, holiday, hopeful, horror, inspiring, love, meditative, melancholic, mellow, melodic, motivational, movie, nature, party, positive, powerful, relaxing, retro, romantic, sad, sexy, slow, soft, soundscape, space, sport, summer, trailer, travel, upbeat, uplifting.

Emotion and theme recognition is a popular task in music information retrieval that is relevant for music search and recommendation systems.

This task involves the prediction of moods and themes conveyed by a music track, given the raw audio. The examples of moods and themes are: happy, dark, epic, melodic, love, film, space etc. The full list is available at: https://github.com/mir-dataset-loaders/mirdata/pull/505 Each track is tagged with at least one tag that serves as a ground-truth.

Acknowledgments

This work was funded by the predoctoral grant MDM-2015-0502-17-2 from the Spanish Ministry of Economy and Competitiveness linked to the Maria de Maeztu Units of Excellence Programme (MDM-2015-0502).

This work has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 765068 "MIP-Frontiers".

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 688382 "AudioCommons".

---

class mirdata.datasets.mtg_jamendo_autotagging_moodtheme.**Dataset**(*data_home=None*, *version='default'*)

The MTG jamendo autotagging moodtheme dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

get_random_track_splits(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

get_track_ids_for_split(*split_number*)

Load a MTG jamendo autotagging moodtheme pre-defined split. There are five different train/validation/tests splits. :Parameters: **split_number** (*int*) – split to be retrieved from 0 to 4

Returns

** dict* – {"train": [...], "validation": [...], "test": [...]} - the train split

Deprecated since version 0.3.6: Use mirdata.datasets.mtg_jamendo_autotagging_moodtheme.get_track_splits

get_track_splits(*split_number=0*)

Get predetermined track splits released alongside this dataset

Parameters
split_number (*int*) – which split split_number to use (0, 1, 2, 3 or 4)

Returns
dict – splits, keyed by split name and with values of lists of track_ids

license()

Print the license

load_audio(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.mtg_jamendo_autotagging_moodtheme.load_audio

load_multitracks()

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
NotImplementedError – If the dataset does not support Multitracks

load_tracks()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

mtrack_ids

Return track ids

Returns
list – A list of track ids

track_ids

Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.mtg_jamendo_autotagging_moodtheme.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

MTG jamendo autotagging moodtheme Track class

Parameters
track_id (*str*) – track id of the track (JAMENDO track id)

Variables
audio_path (*str*) – Path to the audio file

Other Parameters

- **artist_id** (*str*) – JAMENDO artist id
- **album_id** (*str*) – JAMENDO album id
- **duration** (*float*) – track duration
- **tags** (*str*) – autotagging moodtheme annotations

property audio: Tuple[numpy.ndarray, float] | None

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

mirdata.datasets.mtg_jamendo_autotagging_moodtheme.load_audio(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a MTG jamendo autotagging moodtheme audio file.

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to an audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

2.5.34 maestro

MAESTRO Dataset Loader

Dataset Info

MAESTRO (MIDI and Audio Edited for Synchronous TRacks and Organization) is a dataset composed of over 200 hours of virtuosic piano performances captured with fine alignment (~3 ms) between note labels and audio waveforms.

The dataset is created and released by Google's Magenta team.

The dataset contains over 200 hours of paired audio and MIDI recordings from ten years of International Piano-e-Competition. The MIDI data includes key strike velocities and sustain/sostenuto/una corda pedal positions. Audio and MIDI files are aligned with 3 ms accuracy and sliced to individual musical pieces, which are annotated with composer, title, and year of performance. Uncompressed audio is of CD quality or higher (44.1–48 kHz 16-bit PCM stereo).

A train/validation/test split configuration is also proposed, so that the same composition, even if performed by multiple contestants, does not appear in multiple subsets. Repertoire is mostly classical, including composers from the 17th to early 20th century.

The dataset is made available by Google LLC under a Creative Commons Attribution Non-Commercial Share-Alike 4.0 (CC BY-NC-SA 4.0) license.

This loader supports MAESTRO version 2.

For more details, please visit: https://magenta.tensorflow.org/datasets/maestro

---

class mirdata.datasets.maestro.**Dataset**(*data_home=None*, *version='default'*)

The maestro dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns

str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*)

Download the dataset

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.

- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

`get_mtrack_splits()`

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits(splits, seed=42, split_names=None)`

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_random_track_splits(splits, seed=42, split_names=None)`

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_track_splits()`

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- **AttributeError** – If this dataset does not have tracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of track_ids

`license()`

Print the license

`load_audio(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.maestro.load_audio

`load_midi(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.io.load_midi

`load_multitracks()`

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
NotImplementedError – If the dataset does not support Multitracks

`load_notes(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.io.load_notes_from_midi

`load_tracks()`

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns
list – A list of track ids

`track_ids`

Return track ids

Returns
list – A list of track ids

`validate(verbose=True)`

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

`class mirdata.datasets.maestro.Track(track_id, data_home, dataset_name, index, metadata)`

MAESTRO Track class

Parameters
track_id (*str*) – track id of the track

Variables

- **audio_path** (*str*) – Path to the track's audio file

- **canonical_composer** (*str*) – Composer of the piece, standardized on a single spelling for a given name.
- **canonical_title** (*str*) – Title of the piece. Not guaranteed to be standardized to a single representation.
- **duration** (*float*) – Duration in seconds, based on the MIDI file.
- **midi_path** (*str*) – Path to the track's MIDI file
- **split** (*str*) – Suggested train/validation/test split.
- **track_id** (*str*) – track id
- **year** (*int*) – Year of performance.

Cached Property:
midi (pretty_midi.PrettyMIDI): object containing MIDI annotations notes (NoteData): annotated piano notes

property audio: Tuple[numpy.ndarray, float] | None

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.maestro.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a MAESTRO audio file.

Parameters
fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

2.5.35 medley_solos_db

Medley-solos-DB Dataset Loader.

Dataset Info

Medley-solos-DB is a cross-collection dataset for automatic musical instrument recognition in solo recordings. It consists of a training set of 3-second audio clips, which are extracted from the MedleyDB dataset (Bittner et al., ISMIR 2014) as well as a test set of 3-second clips, which are extracted from the solosDB dataset (Essid et al., IEEE TASLP 2009).

Each of these clips contains a single instrument among a taxonomy of eight:

0. clarinet,
1. distorted electric guitar,
2. female singer,
3. flute,
4. piano,
5. tenor saxophone,
6. trumpet, and
7. violin.

The Medley-solos-DB dataset is the dataset that is used in the benchmarks of musical instrument recognition in the publications of Lostanlen and Cella (ISMIR 2016) and Andén et al. (IEEE TSP 2019).

`class mirdata.datasets.medley_solos_db.Dataset(data_home=None, version='default')`

The medley_solos_db dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

`choice_multitrack()`

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_audio`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.medley_solos_db.load_audio

`load_multitracks`()

Load all multitracks in the dataset

Returns

dict – {*mtrack_id*: multitrack data}

Raises

`NotImplementedError` – If the dataset does not support Multitracks

`load_tracks`()

Load all tracks in the dataset

Returns

dict – {*track_id*: track data}

Raises

`NotImplementedError` – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns

list – A list of track ids

`track_ids`

Return track ids

Returns

list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.medley_solos_db.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

medley_solos_db Track class

Parameters
track_id (*str*) – track id of the track

Variables

- **audio_path** (*str*) – path to the track's audio file
- **instrument** (*str*) – instrument encoded by its English name
- **instrument_id** (*int*) – instrument encoded as an integer
- **song_id** (*int*) – song encoded as an integer
- **subset** (*str*) – either equal to 'train', 'validation', or 'test'
- **track_id** (*str*) – track id

property audio: Tuple[numpy.ndarray, float] | None

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.medley_solos_db.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a Medley Solos DB audio file.

Parameters
fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal

- float - The sample rate of the audio file

2.5.36 medleydb_melody

MedleyDB melody Dataset Loader

Dataset Info

MedleyDB melody is a subset of the MedleyDB dataset containing only the mixtures and melody annotations.

MedleyDB is a dataset of annotated, royalty-free multitrack recordings. MedleyDB was curated primarily to support research on melody extraction, addressing important shortcomings of existing collections. For each song we provide melody f0 annotations as well as instrument activations for evaluating automatic instrument recognition.

For more details, please visit: https://medleydb.weebly.com

`class mirdata.datasets.medleydb_melody.Dataset(data_home=None, version='default')`

The medleydb_melody dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

`choice_multitrack()`

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

`choice_track()`

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

`cite()`

Print the reference

`property default_path`

Get the default path for the dataset

Returns

str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

get_random_track_splits(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

get_track_splits()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- **AttributeError** – If this dataset does not have tracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of track_ids

license()

Print the license

load_audio(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.medleydb_melody.load_audio

load_melody(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.medleydb_melody.load_melody

load_melody3(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.medleydb_melody.load_melody3

load_multitracks()

Load all multitracks in the dataset

Returns

dict – {*mtrack_id*: multitrack data}

Raises

NotImplementedError – If the dataset does not support Multitracks

load_tracks()

Load all tracks in the dataset

Returns

dict – {*track_id*: track data}

Raises

NotImplementedError – If the dataset does not support Tracks

mtrack_ids

Return track ids

Returns

list – A list of track ids

track_ids

Return track ids

Returns

list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters

verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally

- list - files which have an invalid checksum

class mirdata.datasets.medleydb_melody.Track(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

medleydb_melody Track class

Parameters

track_id (*str*) – track id of the track

Variables

- **artist** (*str*) – artist
- **audio_path** (*str*) – path to the audio file
- **genre** (*str*) – genre
- **is_excerpt** (*bool*) – True if the track is an excerpt
- **is_instrumental** (*bool*) – True of the track does not contain vocals
- **melody1_path** (*str*) – path to the melody1 annotation file
- **melody2_path** (*str*) – path to the melody2 annotation file
- **melody3_path** (*str*) – path to the melody3 annotation file
- **n_sources** (*int*) – Number of instruments in the track
- **title** (*str*) – title
- **track_id** (*str*) – track id

Other Parameters

- **melody1** (*F0Data*) – the pitch of the single most predominant source (often the voice)
- **melody2** (*F0Data*) – the pitch of the predominant source for each point in time
- **melody3** (*MultiF0Data*) – the pitch of any melodic source. Allows for more than one f0 value at a time

property audio: Tuple[numpy.ndarray, float] | None

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters

key (*string*) – Index key of the audio or annotation type

Returns

str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns

jams.JAMS – the track's data in jams format

mirdata.datasets.medleydb_melody.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a MedleyDB audio file.

Parameters
fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.medleydb_melody.**load_melody**(*fhandle: TextIO*) → *F0Data*

Load a MedleyDB melody1 or melody2 annotation file

Parameters
fhandle (*str or file-like*) – File-like object or path to a melody annotation file

Raises
IOError – if melody_path does not exist

Returns
F0Data – melody data

mirdata.datasets.medleydb_melody.**load_melody3**(*fhandle: TextIO*) → *MultiF0Data*

Load a MedleyDB melody3 annotation file

Parameters
fhandle (*str or file-like*) – File-like object or melody 3 melody annotation path

Raises
IOError – if melody_path does not exist

Returns
MultiF0Data – melody 3 annotation data

2.5.37 medleydb_pitch

MedleyDB pitch Dataset Loader

Dataset Info

MedleyDB Pitch is a pitch-tracking subset of the MedleyDB dataset containing only f0-annotated, monophonic stems.

MedleyDB is a dataset of annotated, royalty-free multitrack recordings. MedleyDB was curated primarily to support research on melody extraction, addressing important shortcomings of existing collections. For each song we provide melody f0 annotations as well as instrument activations for evaluating automatic instrument recognition.

For more details, please visit: https://medleydb.weebly.com

---

class mirdata.datasets.medleydb_pitch.**Dataset**(*data_home=None*, *version='default'*)

The medleydb_pitch dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset

- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits`(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_audio`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.medleydb_pitch.load_audio

`load_multitracks`()

Load all multitracks in the dataset

Returns

dict – {*mtrack_id*: multitrack data}

Raises

`NotImplementedError` – If the dataset does not support Multitracks

`load_notes`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.medleydb_pitch.load_notes

load_pitch(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.medleydb_pitch.load_pitch

load_tracks()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

mtrack_ids

Return track ids

Returns
list – A list of track ids

track_ids

Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.medleydb_pitch.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

medleydb_pitch Track class

Parameters
track_id (*str*) – track id of the track

Variables

- **artist** (*str*) – artist
- **audio_path** (*str*) – path to the audio file
- **genre** (*str*) – genre
- **instrument** (*str*) – instrument of the track
- **notes_pyin_path** (*str*) – path to the pyin note annotation file
- **pitch_path** (*str*) – path to the pitch annotation file
- **title** (*str*) – title
- **track_id** (*str*) – track id

Other Parameters

- **pitch** (*F0Data*) – human annotated pitch
- **notes_pyin** (*NoteData*) – notes estimated by the pyin algorithm. Not available in version 2.0

property audio: Tuple[numpy.ndarray, float] | None

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.medleydb_pitch.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a MedleyDB audio file.

Parameters
fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.medleydb_pitch.**load_notes**(*fhandle: TextIO*) → *NoteData* | None

load a note annotation file

Parameters
fhandle (*str or file-like*) – str or file-like to note annotation file

Raises
IOError – if file doesn't exist

Returns
NoteData – note annotation

mirdata.datasets.medleydb_pitch.**load_pitch**(*fhandle: TextIO*) → *F0Data*

load a MedleyDB pitch annotation file

Parameters
fhandle (*str or file-like*) – str or file-like to pitch annotation file

Raises
IOError – if the path doesn't exist

Returns
F0Data – pitch annotation

2.5.38 mridangam_stroke

Mridangam Stroke Dataset Loader

Dataset Info

The Mridangam Stroke dataset is a collection of individual strokes of the Mridangam in various tonics. The dataset comprises of 10 different strokes played on Mridangams with 6 different tonic values. The audio examples were recorded from a professional Carnatic percussionist in a semi-anechoic studio conditions by Akshay Anantapadmanabhan.

Total audio samples: 6977

Used microphones:

- SM-58 microphones
- H4n ZOOM recorder.

Audio specifications:

- Sampling frequency: 44.1 kHz
- Bit-depth: 16 bit
- Audio format: .wav

The dataset can be used for training models for each Mridangam stroke. The presentation of the dataset took place on the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP 2013) on May 2013. You can read the full publication here: https://repositori.upf.edu/handle/10230/25756

Mridangam Dataset is annotated by storing the informat of each track in their filenames. The structure of the filename is:

```
<TrackID>__<AuthorName>__<StrokeName>-<Tonic>-<InstanceNum>.wav
```

The dataset is made available by CompMusic under a Creative Commons Attribution 3.0 Unported (CC BY 3.0) License.

For more details, please visit: https://compmusic.upf.edu/mridangam-stroke-dataset

class mirdata.datasets.mridangam_stroke.**Dataset**(*data_home=None*, *version='default'*)

The mridangam_stroke dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list

- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_audio`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.mridangam_stroke.load_audio

`load_multitracks`()

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

`load_tracks`()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
`NotImplementedError` – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns
list – A list of track ids

`track_ids`

Return track ids

Returns
list – A list of track ids

`validate`(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

`class mirdata.datasets.mridangam_stroke.Track`(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

Mridangam Stroke track class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored.

Variables

- **track_id** (*str*) – track id
- **audio_path** (*str*) – audio path
- **stroke_name** (*str*) – name of the Mridangam stroke present in Track
- **tonic** (*str*) – tonic of the stroke in the Track

`property audio: Tuple[numpy.ndarray, float] | None`

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

`get_path`(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

`to_jams`()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.mridangam_stroke.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a Mridangam Stroke Dataset audio file.

Parameters
fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

2.5.39 orchset

ORCHSET Dataset Loader

Dataset Info

Orchset is intended to be used as a dataset for the development and evaluation of melody extraction algorithms. This collection contains 64 audio excerpts focused on symphonic music with their corresponding annotation of the melody.

For more details, please visit: https://zenodo.org/record/1289786#.XREpzaeZPx6

class mirdata.datasets.orchset.**Dataset**(*data_home=None*, *version='default'*)

The orchset dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

get_random_track_splits(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list

- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_track_splits()`

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license()`

Print the license

`load_audio_mono(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.orchset.load_audio_mono

`load_audio_stereo(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.orchset.load_audio_stereo

`load_melody(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.orchset.load_melody

`load_multitracks()`

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

`load_tracks()`

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
`NotImplementedError` – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns
list – A list of track ids

`track_ids`

Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.orchset.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

orchset Track class

Parameters
track_id (*str*) – track id of the track

Variables

- **alternating_melody** (*bool*) – True if the melody alternates between instruments
- **audio_path_mono** (*str*) – path to the mono audio file
- **audio_path_stereo** (*str*) – path to the stereo audio file
- **composer** (*str*) – the work's composer
- **contains_brass** (*bool*) – True if the track contains any brass instrument
- **contains_strings** (*bool*) – True if the track contains any string instrument
- **contains_winds** (*bool*) – True if the track contains any wind instrument
- **excerpt** (*str*) – True if the track is an excerpt
- **melody_path** (*str*) – path to the melody annotation file
- **only_brass** (*bool*) – True if the track contains brass instruments only
- **only_strings** (*bool*) – True if the track contains string instruments only
- **only_winds** (*bool*) – True if the track contains wind instruments only
- **predominant_melodic_instruments** (*list*) – List of instruments which play the melody
- **track_id** (*str*) – track id
- **work** (*str*) – The musical work

Other Parameters
melody (*F0Data*) – melody annotation

property audio_mono: Tuple[numpy.ndarray, float] | None

the track's audio (mono)

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

property audio_stereo: Tuple[numpy.ndarray, float] | None

the track's audio (stereo)

Returns

- np.ndarray - the mono audio signal

- float - The sample rate of the audio file

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.orchset.**load_audio_mono**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load an Orchset audio file.

Parameters
fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.orchset.**load_audio_stereo**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load an Orchset audio file.

Parameters
fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the stereo audio signal
- float - The sample rate of the audio file

mirdata.datasets.orchset.**load_melody**(*fhandle: TextIO*) → *F0Data*

Load an Orchset melody annotation file

Parameters
fhandle (*str or file-like*) – File-like object or path to melody annotation file

Raises
IOError – if melody_path doesn't exist

Returns
F0Data – melody annotation data

2.5.40 phenicx_anechoic

PHENICX-Anechoic Dataset Loader

Dataset Info

This dataset includes audio and annotations useful for tasks as score-informed source separation, score following, multi-pitch estimation, transcription or instrument detection, in the context of symphonic music: M. Miron, J. Carabias-Orti, J. J. Bosch, E. Gómez and J. Janer, "Score-informed source separation for multi-channel orchestral recordings", Journal of Electrical and Computer Engineering (2016))"

We do not provide the original audio files, which can be found at the web page hosted by Aalto University. However, with their permission we distribute the denoised versions for some of the anechoic orchestral recordings. The original dataset was introduced in: Pätynen, J., Pulkki, V., and Lokki, T., "Anechoic recording system for symphony orchestra," Acta Acustica united with Acustica, vol. 94, nr. 6, pp. 856-865, November/December 2008.

Additionally, we provide the associated musical note onset and offset annotations, and the Roomsim configuration files used to generate the multi-microphone recordings.

The original anechoic dataset in Pätynen et al. consists of four passages of symphonic music from the Classical and Romantic periods. This work presented a set of anechoic recordings for each of the instruments, which were then synchronized between them so that they could later be combined to a mix of the orchestra. In order to keep the evaluation setup consistent between the four pieces, we selected the following instruments: violin, viola, cello, double bass, oboe, flute, clarinet, horn, trumpet and bassoon. A list of the characteristics of the four pieces can be found below:

Mozart - duration: 3min 47s - period: classical - no. sources: 8 - total no. instruments: 10 - max. instruments/source: 2

Beethoven - duration: 3min 11s - period: classical - no. sources: 10 - total no. instruments: 20 - max. instruments/source: 4

Beethoven - duration: 2min 12s - period: romantic - no. sources: 10 - total no. instruments: 30 - max. instruments/source: 4

Bruckner - duration: 1min 27s - period: romantic - no. sources: 10 - total no. instruments: 39 - max. instruments/source: 12

For more details, please visit: https://www.upf.edu/web/mtg/phenicx-anechoic

---

`class mirdata.datasets.phenicx_anechoic.Dataset(data_home=None, version='default')`

The Phenicx-Anechoic dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list

- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_audio`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.phenicx_anechoic.load_audio

`load_multitracks`()

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

`load_score`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.phenicx_anechoic.load_score

`load_tracks`()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
`NotImplementedError` – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns
list – A list of track ids

`track_ids`

Return track ids

Returns
list – A list of track ids

`validate`(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

`class mirdata.datasets.phenicx_anechoic.MultiTrack`(*mtrack_id*, *data_home*, *dataset_name*, *index*, *track_class*, *metadata*)

Phenicx-Anechoic MultiTrack class

Parameters

- **mtrack_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored. If *None*, looks for the data in the default directory, *~/mir_datasets/Phenicx-Anechoic*

Variables

- **track_audio_property** (*str*) – the attribute of track which is used for mixing
- **mtrack_id** (*str*) – multitrack id
- **piece** (*str*) – the classical music piece associated with this multitrack
- **tracks** (*dict*) – dict of track ids and the corresponding Tracks
- **instruments** (*dict*) – dict of instruments and the corresponding track
- **sections** (*dict*) – dict of sections and the corresponding list of tracks for each section

`get_audio_for_instrument`(*instrument*)

Get the audio for a particular instrument

Parameters
instrument (*str*) – the instrument to get audio for

Returns
np.ndarray – instrument audio with shape (n_samples, n_channels)

`get_audio_for_section`(*section*)

Get the audio for a particular section

Parameters
section (*str*) – the section to get audio for

Returns
np.ndarray – section audio with shape (n_samples, n_channels)

`get_mix()`

Create a linear mixture given a subset of tracks.

Parameters
track_keys (*list*) – list of track keys to mix together

Returns
np.ndarray – mixture audio with shape (n_samples, n_channels)

`get_notes_for_instrument(instrument, notes_property='notes')`

Get the notes for a particular instrument

Parameters

- **instrument** (*str*) – the instrument to get the notes for
- **notes_property** (*str*) – the attribute associated with NoteData, notes or notes_original

Returns
NoteData – Note data for the instrument

`get_notes_for_section(section, notes_property='notes')`

Get the notes for a particular section

Parameters

- **section** (*str*) – the section to get the notes for
- **notes_property** (*str*) – the attribute associated with NoteData, notes or notes_original

Returns
NoteData – Note data for the section

`get_notes_target(track_keys, notes_property='notes')`

Get the notes for all the tracks

Parameters

- **track_keys** (*list*) – list of track keys to get the NoteData for
- **notes_property** (*str*) – the attribute associated with NoteData, notes or notes_original

Returns
NoteData – Note data for the tracks

`get_path(key)`

Get absolute path to multitrack audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

`get_random_target(n_tracks=None, min_weight=0.3, max_weight=1.0)`

Get a random target by combining a random selection of tracks with random weights

Parameters

- **n_tracks** (*int or None*) – number of tracks to randomly mix. If None, uses all tracks
- **min_weight** (*float*) – minimum possible weight when mixing

- **max_weight** (*float*) – maximum possible weight when mixing

Returns

- np.ndarray - mixture audio with shape (n_samples, n_channels)
- list - list of keys of included tracks
- list - list of weights used to mix tracks

`get_target`(*track_keys*, *weights=None*, *average=True*, *enforce_length=True*)

Get target which is a linear mixture of tracks

Parameters

- **track_keys** (*list*) – list of track keys to mix together
- **weights** (*list or None*) – list of positive scalars to be used in the average
- **average** (*bool*) – if True, computes a weighted average of the tracks if False, computes a weighted sum of the tracks
- **enforce_length** (*bool*) – If True, raises ValueError if the tracks are not the same length. If False, pads audio with zeros to match the length of the longest track

Returns

np.ndarray – target audio with shape (n_channels, n_samples)

Raises

`ValueError` – if sample rates of the tracks are not equal if enforce_length=True and lengths are not equal

`class mirdata.datasets.phenicx_anechoic.Track`(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

Phenicx-Anechoic Track class

Parameters

track_id (*str*) – track id of the track

Variables

- **audio_path** (*list*) – path to the audio files
- **notes_path** (*list*) – path to the score files
- **notes_original_path** (*list*) – path to the original score files
- **instrument** (*str*) – the name of the instrument
- **piece** (*str*) – the name of the piece
- **n_voices** (*int*) – the number of voices in this instrument
- **track_id** (*str*) – track id

Other Parameters

- **notes** (*NoteData*) – notes annotations that have been time-aligned to the audio
- **notes_original** (*NoteData*) – original score representation, not time-aligned

`property audio: Tuple[numpy.ndarray, float] | None`

the track's audio

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

get_audio_voice(*id_voice: int*) → Tuple[numpy.ndarray, float] | None

the track's audio

Parameters
id_voice (*int*) – The integer identifier for the voice e.g. 2 for bassoon-2

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

notes

the track's notes corresponding to the score aligned to the audio

Returns
NoteData – Note data for the track

notes_original

the track's notes corresponding to the original score

Returns
NoteData – Note data for the track

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.phenicx_anechoic.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a Phenicx-Anechoic audio file.

Parameters
fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the audio signal
- float - The sample rate of the audio file

mirdata.datasets.phenicx_anechoic.**load_score**(*fhandle: TextIO*) → *NoteData*

Load a Phenicx-Anechoic score file.

Parameters
fhandle (*str or file-like*) – File-like object or path to score file

Returns
NoteData – Note data for the given track

2.5.41 queen

Queen Dataset Loader

Dataset Info

Queen Dataset includes chord, key, and segmentation annotations for 51 Queen songs. Details can be found in http://matthiasmauch.net/_pdf/mauch_omp_2009.pdf and http://isophonics.net/content/reference-annotations-queen.

The CDs used in this dataset are: Queen: Greatest Hits I, Parlophone, 0777 7 8950424 Queen: Greatest Hits II, Parlophone, CDP 7979712 Queen: Greatest Hits III, Parlophone, 7243 52389421

In the progress of labelling the chords, C4DM researchers used the following literature to verify their judgements:

Queen, Greatest Hits I, International Music Publications Ltd, London, ISBN 0-571-52828-7

Queen, Greatest Hits II, Queen Music Ltd./EMI Music Publishing (Barnes Music Engraving), ISBN 0-86175-465-4

Acknowledgements We'd like to thank our student annotators:

Eric Gyingy Diako Rasoul Felix Stiller Helena du Toit Vinh Ton Chuks Chiejine

---

class mirdata.datasets.queen.**Dataset**(*data_home=None*, *version='default'*)

Queen dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

load_audio(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.queen.load_audio

load_chords(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.queen.load_chords

load_key(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.queen.load_key

load_sections(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.queen.load_sections

class mirdata.datasets.queen.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

Queen track class

Parameters

track_id (*str*) – track id of the track

Variables

- **audio_path** (*str*) – track audio path
- **chords_path** (*str*) – chord annotation path
- **keys_path** (*str*) – key annotation path
- **sections_path** (*str*) – sections annotation path
- **title** (*str*) – title of the track
- **track_id** (*str*) – track id

Other Parameters

- **chords** (*ChordData*) – human-labeled chord annotations
- **key** (*KeyData*) – local key annotations
- **sections** (*SectionData*) – section annotations

property audio: Tuple[numpy.ndarray, float] | None

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

to_jams()

the track's data in jams format

Returns

jams.JAMS – return track data in jam format

mirdata.datasets.queen.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a Queen audio file.

Parameters

fhandle (*str*) – path to an audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.queen.**load_chords**(*fhandle: TextIO*) → *ChordData*

Load Queen format chord data from a file

Parameters

fhandle (*str or file-like*) – path or file-like object pointing to a chord file

Returns

(ChordData) – loaded chord data

mirdata.datasets.queen.**load_key**(*fhandle: TextIO*) → *KeyData*

Load Queen format key data from a file

Parameters

fhandle (*str or file-like*) – path or file-like object pointing to a key file

Returns

(KeyData) – loaded key data

mirdata.datasets.queen.**load_sections**(*fhandle: TextIO*) → *SectionData*

Load Queen format section data from a file

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to a section file

Returns
(SectionData) – loaded section data

2.5.42 rwc_classical

RWC Classical Dataset Loader

Dataset Info

The Classical Music Database consists of 50 pieces

- Symphonies: 4 pieces
- Concerti: 2 pieces
- Orchestral music: 4 pieces
- Chamber music: 10 pieces
- Solo performances: 24 pieces
- Vocal performances: 6 pieces

A note about the Beat annotations:

- 48 corresponds to the duration of a quarter note (crotchet)
- 24 corresponds to the duration of an eighth note (quaver)
- 384 corresponds to the position of a downbeat

In 4/4 time signature, they correspond as follows:

```
384: 1st beat in a measure (i.e., downbeat position)
48: 2nd beat
96: 3rd beat
144 4th beat
```

In 3/4 time signature, they correspond as follows:

```
384: 1st beat in a measure (i.e., downbeat position)
48: 2nd beat
96: 3rd beat
```

In 6/8 time signature, they correspond as follows:

```
384: 1st beat in a measure (i.e., downbeat position)
24: 2nd beat
48: 3rd beat
72: 4th beat
96: 5th beat
120: 6th beat
```

For more details, please visit: https://staff.aist.go.jp/m.goto/RWC-MDB/rwc-mdb-c.html

class mirdata.datasets.rwc_classical.Dataset(*data_home=None*, *version='default'*)

The rwc_classical dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns

str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download

- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

get_random_track_splits(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

get_track_splits()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- **AttributeError** – If this dataset does not have tracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of track_ids

license()

Print the license

load_audio(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.rwc_classical.load_audio

load_beats(*args, **kwargs)

Deprecated since version 0.3.4: Use mirdata.datasets.rwc_classical.load_beats

load_multitracks()

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
NotImplementedError – If the dataset does not support Multitracks

load_sections(*args, **kwargs)

Deprecated since version 0.3.4: Use mirdata.datasets.rwc_classical.load_sections

load_tracks()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

mtrack_ids

Return track ids

Returns
list – A list of track ids

track_ids

Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.rwc_classical.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

rwc_classical Track class

Parameters
track_id (*str*) – track id of the track

Variables

- **artist** (*str*) – the track's artist
- **audio_path** (*str*) – path of the audio file
- **beats_path** (*str*) – path of the beat annotation file
- **category** (*str*) – One of 'Symphony', 'Concerto', 'Orchestral', 'Solo', 'Chamber', 'Vocal', or blank.

- **composer** (*str*) – Composer of this Track.
- **duration** (*float*) – Duration of the track in seconds
- **piece_number** (*str*) – Piece number of this Track, [1-50]
- **sections_path** (*str*) – path of the section annotation file
- **suffix** (*str*) – string within M01-M06
- **title** (*str*) – Title of The track.
- **track_id** (*str*) – track id
- **track_number** (*str*) – CD track number of this Track

Other Parameters

- **sections** (*SectionData*) – human-labeled section annotations
- **beats** (*BeatData*) – human-labeled beat annotations

property audio: Tuple[numpy.ndarray, float] | None

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.rwc_classical.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a RWC audio file.

Parameters
fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.rwc_classical.**load_beats**(*fhandle: TextIO*) → *BeatData*

Load rwc beat data from a file

Parameters
fhandle (*str or file-like*) – File-like object or path to beats annotation file

Returns
BeatData – beat data

mirdata.datasets.rwc_classical.**load_sections**(*fhandle: TextIO*) → *SectionData* | None

Load rwc section data from a file

Parameters
fhandle (*str or file-like*) – File-like object or path to sections annotation file

Returns
SectionData – section data

2.5.43 rwc_jazz

RWC Jazz Dataset Loader.

Dataset Info

The Jazz Music Database consists of 50 pieces:

- **Instrumentation variations:** 35 pieces (5 pieces × 7 instrumentations).

  The instrumentation-variation pieces were recorded to obtain different versions of the same piece; i.e., different arrangements performed by different player instrumentations. Five standard-style jazz pieces were originally composed and then performed in modern-jazz style using the following seven instrumentations:

  1. Piano solo
  2. Guitar solo
  3. Duo: Vibraphone + Piano, Flute + Piano, and Piano + Bass
  4. Piano trio: Piano + Bass + Drums
  5. Piano trio + Trumpet or Tenor saxophone
  6. Octet: Piano trio + Guitar + Alto saxophone + Baritone saxophone + Tenor saxophone × 2
  7. Piano trio + Vibraphone or Flute

- **Style variations:** 9 pieces

  The style-variation pieces were recorded to represent various styles of jazz. They include four well-known public-domain pieces and consist of

  1. Vocal jazz: 2 pieces (including "Aura Lee")
  2. Big band jazz: 2 pieces (including "The Entertainer")
  3. Modal jazz: 2 pieces
  4. Funky jazz: 2 pieces (including "Silent Night")
  5. Free jazz: 1 piece (including "Joyful, Joyful, We Adore Thee")

- **Fusion (crossover):** 6 pieces

  The fusion pieces were recorded to obtain music that combines elements of jazz with other styles such as popular, rock, and latin. They include music with an eighth-note feel, music with a sixteenth-note feel, and Latin jazz music.

For more details, please visit: https://staff.aist.go.jp/m.goto/RWC-MDB/rwc-mdb-j.html

---

class mirdata.datasets.rwc_jazz.Dataset(*data_home=None*, *version='default'*)

The rwc_jazz dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns

str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

`get_mtrack_splits()`

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- `AttributeError` – If this dataset does not have multitracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits(splits, seed=42, split_names=None)`

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_random_track_splits(splits, seed=42, split_names=None)`

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_track_splits()`

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of track_ids

`license()`

Print the license

`load_audio(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.rwc_jazz.load_audio

`load_beats(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.rwc_jazz.load_beats

load_multitracks()

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
NotImplementedError – If the dataset does not support Multitracks

load_sections(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.rwc_jazz.load_sections

load_tracks()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

mtrack_ids

Return track ids

Returns
list – A list of track ids

track_ids

Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.rwc_jazz.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

rwc_jazz Track class

Parameters
track_id (*str*) – track id of the track

Variables

- **artist** (*str*) – Artist name
- **audio_path** (*str*) – path of the audio file
- **beats_path** (*str*) – path of the beat annotation file
- **duration** (*float*) – Duration of the track in seconds
- **instruments** (*str*) – list of used instruments.
- **piece_number** (*str*) – Piece number of this Track, [1-50]

- **sections_path** (*str*) – path of the section annotation file
- **suffix** (*str*) – M01-M04
- **title** (*str*) – Title of The track.
- **track_id** (*str*) – track id
- **track_number** (*str*) – CD track number of this Track
- **variation** (*str*) – style variations

Other Parameters

- **sections** (*SectionData*) – human-labeled section data
- **beats** (*BeatData*) – human-labeled beat data

property audio: Tuple[numpy.ndarray, float] | None

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

2.5.44 rwc_popular

RWC Popular Dataset Loader

Dataset Info

The Popular Music Database consists of 100 songs — 20 songs with English lyrics performed in the style of popular music typical of songs on the American hit charts in the 1980s, and 80 songs with Japanese lyrics performed in the style of modern Japanese popular music typical of songs on the Japanese hit charts in the 1990s.

For more details, please visit: https://staff.aist.go.jp/m.goto/RWC-MDB/rwc-mdb-p.html

---

class mirdata.datasets.rwc_popular.**Dataset**(*data_home=None*, *version='default'*)

The rwc_popular dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –

- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns

str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits`(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_audio`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.rwc_popular.load_audio

`load_beats`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.rwc_popular.load_beats

`load_chords`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.rwc_popular.load_chords

`load_multitracks`()

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
NotImplementedError – If the dataset does not support Multitracks

load_sections(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.rwc_popular.load_sections

load_tracks()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

load_vocal_activity(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.rwc_popular.load_vocal_activity

mtrack_ids

Return track ids

Returns
list – A list of track ids

track_ids

Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.rwc_popular.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

rwc_popular Track class

Parameters
track_id (*str*) – track id of the track

Variables

- **artist** (*str*) – artist
- **audio_path** (*str*) – path of the audio file
- **beats_path** (*str*) – path of the beat annotation file
- **chords_path** (*str*) – path of the chord annotation file
- **drum_information** (*str*) – If the drum is 'Drum sequences', 'Live drums', or 'Drum loops'
- **duration** (*float*) – Duration of the track in seconds
- **instruments** (*str*) – List of used instruments
- **piece_number** (*str*) – Piece number, [1-50]

- **sections_path** (*str*) – path of the section annotation file
- **singer_information** (*str*) – could be male, female or vocal group
- **suffix** (*str*) – M01-M04
- **tempo** (*str*) – Tempo of the track in BPM
- **title** (*str*) – title
- **track_id** (*str*) – track id
- **track_number** (*str*) – CD track number
- **voca_inst_path** (*str*) – path of the vocal/instrumental annotation file

Other Parameters

- **sections** (*SectionData*) – human-labeled section annotation
- **beats** (*BeatData*) – human-labeled beat annotation
- **chords** (*ChordData*) – human-labeled chord annotation
- **vocal_instrument_activity** (*EventData*) – human-labeled vocal/instrument activity

property audio: Tuple[numpy.ndarray, float] | None

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.rwc_popular.**load_chords**(*fhandle: TextIO*) → *ChordData*

Load rwc chord data from a file

Parameters
fhandle (*str or file-like*) – File-like object or path to chord annotation file

Returns
ChordData – chord data

mirdata.datasets.rwc_popular.**load_vocal_activity**(*fhandle: TextIO*) → *EventData*

Load rwc vocal activity data from a file

Parameters
fhandle (*str or file-like*) – File-like object or path to vocal activity annotation file

Returns
EventData – vocal activity data

2.5.45 salami

SALAMI Dataset Loader

Dataset Info

The SALAMI dataset contains Structural Annotations of a Large Amount of Music Information: the public portion contains over 2200 annotations of over 1300 unique tracks.

NB: mirdata relies on the **corrected** version of the 2.0 annotations: Details can be found at https://github.com/bmcfee/salami-data-public/tree/hierarchy-corrections and https://github.com/DDMAL/salami-data-public/pull/15.

For more details, please visit: https://github.com/DDMAL/salami-data-public

`class mirdata.datasets.salami.Dataset(data_home=None, version='default')`

The salami dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

`choice_multitrack()`

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

`choice_track()`

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

`cite()`

Print the reference

`property default_path`

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

get_random_track_splits(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_track_splits()`

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of track_ids

`license()`

Print the license

`load_audio(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.salami.load_audio

`load_multitracks()`

Load all multitracks in the dataset

Returns

dict – {*mtrack_id*: multitrack data}

Raises

`NotImplementedError` – If the dataset does not support Multitracks

`load_sections(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.salami.load_sections

`load_tracks()`

Load all tracks in the dataset

Returns

dict – {*track_id*: track data}

Raises

`NotImplementedError` – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns

list – A list of track ids

`track_ids`

Return track ids

Returns

list – A list of track ids

`validate(verbose=True)`

Validate if the stored dataset is a valid version

Parameters

verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.salami.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

salami Track class

Parameters

track_id (*str*) – track id of the track

Variables

- **annotator_1_id** (*str*) – number that identifies annotator 1
- **annotator_1_time** (*str*) – time that the annotator 1 took to complete the annotation
- **annotator_2_id** (*str*) – number that identifies annotator 1
- **annotator_2_time** (*str*) – time that the annotator 1 took to complete the annotation
- **artist** (*str*) – song artist
- **audio_path** (*str*) – path to the audio file
- **broad_genre** (*str*) – broad genre of the song
- **duration** (*float*) – duration of song in seconds
- **genre** (*str*) – genre of the song
- **sections_annotator1_lowercase_path** (*str*) – path to annotations in hierarchy level 1 from annotator 1
- **sections_annotator1_uppercase_path** (*str*) – path to annotations in hierarchy level 0 from annotator 1
- **sections_annotator2_lowercase_path** (*str*) – path to annotations in hierarchy level 1 from annotator 2
- **sections_annotator2_uppercase_path** (*str*) – path to annotations in hierarchy level 0 from annotator 2
- **source** (*str*) – dataset or source of song
- **title** (*str*) – title of the song

Other Parameters

- **sections_annotator_1_uppercase** (*SectionData*) – annotations in hierarchy level 0 from annotator 1
- **sections_annotator_1_lowercase** (*SectionData*) – annotations in hierarchy level 1 from annotator 1
- **sections_annotator_2_uppercase** (*SectionData*) – annotations in hierarchy level 0 from annotator 2
- **sections_annotator_2_lowercase** (*SectionData*) – annotations in hierarchy level 1 from annotator 2

property **audio**: Tuple[numpy.ndarray, float]

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.salami.**load_audio**(*fpath: str*) → Tuple[numpy.ndarray, float]

Load a Salami audio file.

Parameters
fpath (*str*) – path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.salami.**load_sections**(*fhandle: TextIO*) → *SectionData*

Load salami sections data from a file

Parameters
fhandle (*str or file-like*) – File-like object or path to section annotation file

Returns
SectionData – section data

2.5.46 saraga_carnatic

Saraga Dataset Loader

Dataset Info

This dataset contains time aligned melody, rhythm and structural annotations of Carnatic Music tracks, extracted from the large open Indian Art Music corpora of CompMusic.

The dataset contains the following manual annotations referring to audio files:

- Section and tempo annotations stored as start and end timestamps together with the name of the section and tempo during the section (in a separate file)
- Sama annotations referring to rhythmic cycle boundaries stored as timestamps.
- Phrase annotations stored as timestamps and transcription of the phrases using solfège symbols ({S, r, R, g, G, m, M, P, d, D, n, N}).
- Audio features automatically extracted and stored: pitch and tonic.
- The annotations are stored in text files, named as the audio filename but with the respective extension at the end, for instance: "Bhuvini Dasudane.tempo-manual.txt".

The dataset contains a total of 249 tracks. A total of 168 tracks have multitrack audio.

The files of this dataset are shared with the following license: Creative Commons Attribution Non Commercial Share Alike 4.0 International

Dataset compiled by: Bozkurt, B.; Srinivasamurthy, A.; Gulati, S. and Serra, X.

For more information about the dataset as well as IAM and annotations, please refer to: https://mtg.github.io/saraga/, where a really detailed explanation of the data and annotations is published.

---

class mirdata.datasets.saraga_carnatic.**Dataset**(*data_home=None*, *version='default'*)

The saraga_carnatic dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns

str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

`get_mtrack_splits()`

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits(splits, seed=42, split_names=None)`

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_random_track_splits(splits, seed=42, split_names=None)`

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_track_splits()`

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- **AttributeError** – If this dataset does not have tracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of track_ids

license()

Print the license

load_audio(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.saraga_carnatic.load_audio

load_metadata(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.saraga_carnatic.load_metadata

load_multitracks()

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
NotImplementedError – If the dataset does not support Multitracks

load_phrases(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.saraga_carnatic.load_phrases

load_pitch(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.saraga_carnatic.load_pitch

load_sama(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.saraga_carnatic.load_sama

load_sections(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.saraga_carnatic.load_sections

load_tempo(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.saraga_carnatic.load_tempo

load_tonic(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.saraga_carnatic.load_tonic

load_tracks()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

mtrack_ids

Return track ids

Returns
list – A list of track ids

track_ids

Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.saraga_carnatic.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

Saraga Track Carnatic class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored. default=None If *None*, looks for the data in the default directory, *~/mir_datasets*

Variables

- **audio_path** (*str*) – path to audio file
- **audio_ghatam_path** (*str*) – path to ghatam audio file
- **audio_mridangam_left_path** (*str*) – path to mridangam left audio file
- **audio_mridangam_right_path** (*str*) – path to mridangam right audio file
- **audio_violin_path** (*str*) – path to violin audio file
- **audio_vocal_s_path** (*str*) – path to vocal s audio file
- **audio_vocal_pat** (*str*) – path to vocal pat audio file
- **ctonic_path** (*srt*) – path to ctonic annotation file
- **pitch_path** (*srt*) – path to pitch annotation file
- **pitch_vocal_path** (*srt*) – path to vocal pitch annotation file
- **tempo_path** (*srt*) – path to tempo annotation file
- **sama_path** (*srt*) – path to sama annotation file
- **sections_path** (*srt*) – path to sections annotation file
- **phrases_path** (*srt*) – path to phrases annotation file
- **metadata_path** (*srt*) – path to metadata file

Other Parameters

- **tonic** (*float*) – tonic annotation
- **pitch** (*F0Data*) – pitch annotation
- **pitch_vocal** (*F0Data*) – vocal pitch annotation
- **tempo** (*dict*) – tempo annotations
- **sama** (*BeatData*) – sama section annotations
- **sections** (*SectionData*) – track section annotations
- **phrases** (*SectionData*) – phrase annotations
- **metadata** (*dict*) – track metadata with the following fields:
  - title (str): Title of the piece in the track

- mbid (str): MusicBrainz ID of the track
- album_artists (list, dicts): list of dicts containing the album artists present in the track and its mbid
- artists (list, dicts): list of dicts containing information of the featuring artists in the track
- raaga (list, dict): list of dicts containing information about the raagas present in the track
- form (list, dict): list of dicts containing information about the forms present in the track
- work (list, dicts): list of dicts containing the work present in the piece, and its mbid
- taala (list, dicts): list of dicts containing the talas present in the track and its uuid
- concert (list, dicts): list of dicts containing the concert where the track is present and its mbid

property audio

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.saraga_carnatic.**load_audio**(*audio_path*)

Load a Saraga Carnatic audio file.

Parameters
audio_path (*str*) – path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.saraga_carnatic.**load_metadata**(*fhandle*)

Load a Saraga Carnatic metadata file

Parameters
fhandle (*str or file-like*) – File-like object or path to metadata json

Returns

dict –

metadata with the following fields

- title (str): Title of the piece in the track
- mbid (str): MusicBrainz ID of the track
- album_artists (list, dicts): list of dicts containing the album artists present in the track and its mbid
- artists (list, dicts): list of dicts containing information of the featuring artists in the track
- raaga (list, dict): list of dicts containing information about the raagas present in the track
- form (list, dict): list of dicts containing information about the forms present in the track
- work (list, dicts): list of dicts containing the work present in the piece, and its mbid
- taala (list, dicts): list of dicts containing the talas present in the track and its uuid
- concert (list, dicts): list of dicts containing the concert where the track is present and its mbid

mirdata.datasets.saraga_carnatic.**load_phrases**(*fhandle*)

Load phrases

Parameters
fhandle (*str or file-like*) – Local path where the phrase annotation is stored.

Returns
EventData – phrases annotation for track

mirdata.datasets.saraga_carnatic.**load_pitch**(*fhandle*)

Load pitch

Parameters
fhandle (*str or file-like*) – Local path where the pitch annotation is stored.

Returns
F0Data – pitch annotation

mirdata.datasets.saraga_carnatic.**load_sama**(*fhandle*)

Load sama

Parameters
fhandle (*str or file-like*) – Local path where the sama annotation is stored.

Returns
BeatData – sama annotations

mirdata.datasets.saraga_carnatic.**load_sections**(*fhandle*)

Load sections from carnatic collection

Parameters
fhandle (*str or file-like*) – Local path where the section annotation is stored.

Returns
SectionData – section annotations for track

mirdata.datasets.saraga_carnatic.**load_tempo**(*fhandle*)

Load tempo from carnatic collection

Parameters
fhandle (*str or file-like*) – Local path where the tempo annotation is stored.

Returns

dict –

Dictionary of tempo information with the following keys:

- tempo_apm: tempo in aksharas per minute (APM)
- tempo_bpm: tempo in beats per minute (BPM)
- sama_interval: median duration (in seconds) of one tāla cycle
- beats_per_cycle: number of beats in one cycle of the tāla
- subdivisions: number of aksharas per beat of the tāla

mirdata.datasets.saraga_carnatic.**load_tonic**(*fhandle*)

Load track absolute tonic

Parameters
fhandle (*str or file-like*) – Local path where the tonic path is stored.

Returns
int – Tonic annotation in Hz

2.5.47 saraga_hindustani

Saraga Dataset Loader

Dataset Info

This dataset contains time aligned melody, rhythm and structural annotations of Hindustani Music tracks, extracted from the large open Indian Art Music corpora of CompMusic.

The dataset contains the following manual annotations referring to audio files:

- Section and tempo annotations stored as start and end timestamps together with the name of the section and tempo during the section (in a separate file)
- Sama annotations referring to rhythmic cycle boundaries stored as timestamps
- Phrase annotations stored as timestamps and transcription of the phrases using solfège symbols ({S, r, R, g, G, m, M, P, d, D, n, N})
- Audio features automatically extracted and stored: pitch and tonic.
- The annotations are stored in text files, named as the audio filename but with the respective extension at the end, for instance: "Bhuvini Dasudane.tempo-manual.txt".

The dataset contains a total of 108 tracks.

The files of this dataset are shared with the following license: Creative Commons Attribution Non Commercial Share Alike 4.0 International

Dataset compiled by: Bozkurt, B.; Srinivasamurthy, A.; Gulati, S. and Serra, X.

For more information about the dataset as well as IAM and annotations, please refer to: https://mtg.github.io/saraga/, where a really detailed explanation of the data and annotations is published.

---

class mirdata.datasets.saraga_hindustani.**Dataset**(*data_home=None*, *version='default'*)

The saraga_hindustani dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –

- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns

str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits`(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_audio`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.saraga_hindustani.load_audio

`load_multitracks`()

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

load_phrases(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.saraga_hindustani.load_phrases

load_pitch(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.saraga_hindustani.load_pitch

load_sama(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.saraga_hindustani.load_sama

load_sections(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.saraga_hindustani.load_sections

load_tempo(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.saraga_hindustani.load_tempo

load_tonic(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.saraga_hindustani.load_tonic

load_tracks()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

mtrack_ids

Return track ids

Returns
list – A list of track ids

track_ids

Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.saraga_hindustani.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

Saraga Hindustani Track class

Parameters

- **track_id** (*str*) – track id of the track
- **data_home** (*str*) – Local path where the dataset is stored. default=None If *None*, looks for the data in the default directory, *~/mir_datasets*

Variables

- **audio_path** (*str*) – path to audio file
- **ctonic_path** (*str*) – path to ctonic annotation file
- **pitch_path** (*str*) – path to pitch annotation file
- **tempo_path** (*str*) – path to tempo annotation file
- **sama_path** (*str*) – path to sama annotation file
- **sections_path** (*str*) – path to sections annotation file
- **phrases_path** (*str*) – path to phrases annotation file
- **metadata_path** (*str*) – path to metadata annotation file

Other Parameters

- **tonic** (*float*) – tonic annotation
- **pitch** (*F0Data*) – pitch annotation
- **tempo** (*dict*) – tempo annotations
- **sama** (*BeatData*) – Sama section annotations
- **sections** (*SectionData*) – track section annotations
- **phrases** (*EventData*) – phrase annotations
- **metadata** (*dict*) – track metadata with the following fields
  - title (str): Title of the piece in the track
  - mbid (str): MusicBrainz ID of the track
  - album_artists (list, dicts): list of dicts containing the album artists present in the track and its mbid
  - artists (list, dicts): list of dicts containing information of the featuring artists in the track
  - raags (list, dict): list of dicts containing information about the raags present in the track
  - forms (list, dict): list of dicts containing information about the forms present in the track
  - release (list, dicts): list of dicts containing information of the release where the track is found
  - works (list, dicts): list of dicts containing the work present in the piece, and its mbid
  - taals (list, dicts): list of dicts containing the taals present in the track and its uuid
  - layas (list, dicts): list of dicts containing the layas present in the track and its uuid

property audio

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters

key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

`to_jams()`
Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

`mirdata.datasets.saraga_hindustani.load_audio(audio_path)`
Load a Saraga Hindustani audio file.

Parameters
audio_path (*str*) – path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

`mirdata.datasets.saraga_hindustani.load_metadata(fhandle)`
Load a Saraga Hindustani metadata file

Parameters
fhandle (*str or file-like*) – path to metadata json file

Returns

dict –

metadata with the following fields

- title (str): Title of the piece in the track
- mbid (str): MusicBrainz ID of the track
- album_artists (list, dicts): list of dicts containing the album artists present in the track and its mbid
- artists (list, dicts): list of dicts containing information of the featuring artists in the track
- raags (list, dict): list of dicts containing information about the raags present in the track
- forms (list, dict): list of dicts containing information about the forms present in the track
- release (list, dicts): list of dicts containing information of the release where the track is found
- works (list, dicts): list of dicts containing the work present in the piece, and its mbid
- taals (list, dicts): list of dicts containing the taals present in the track and its uuid
- layas (list, dicts): list of dicts containing the layas present in the track and its uuid

`mirdata.datasets.saraga_hindustani.load_phrases(fhandle)`
Load phrases

Parameters
fhandle (*str or file-like*) – Local path where the phrase annotation is stored. If *None*, returns None.

Returns
EventData – phrases annotation for track

mirdata.datasets.saraga_hindustani.**load_pitch**(*fhandle*)

Load automatic extracted pitch or melody

Parameters
fhandle (*str or file-like*) – Local path where the pitch annotation is stored. If *None*, returns None.

Returns
F0Data – pitch annotation

mirdata.datasets.saraga_hindustani.**load_sama**(*fhandle*)

Load sama

Parameters
fhandle (*str or file-like*) – Local path where the sama annotation is stored. If *None*, returns None.

Returns
SectionData – sama annotations

mirdata.datasets.saraga_hindustani.**load_sections**(*fhandle*)

Load tracks sections

Parameters
fhandle (*str or file-like*) – Local path where the section annotation is stored.

Returns
SectionData – section annotations for track

mirdata.datasets.saraga_hindustani.**load_tempo**(*fhandle*)

Load tempo from hindustani collection

Parameters
fhandle (*str or file-like*) – Local path where the tempo annotation is stored.

Returns

dict – Dictionary of tempo information with the following keys:

- tempo: median tempo for the section in mātrās per minute (MPM)
- matra_interval: tempo expressed as the duration of the mātra (essentially dividing 60 by tempo, expressed in seconds)
- sama_interval: median duration of one tāl cycle in the section
- matras_per_cycle: indicator of the structure of the tāl, showing the number of mātrā in a cycle of the tāl of the recording
- start_time: start time of the section
- duration: duration of the section

mirdata.datasets.saraga_hindustani.**load_tonic**(*fhandle*)

Load track absolute tonic

Parameters
fhandle (*str or file-like*) – Local path where the tonic path is stored. If *None*, returns None.

Returns
int – Tonic annotation in Hz

2.5.48 scms

Saraga-Carnatic-Melody-Synth loader

Dataset Info

This dataset contains time aligned vocal melody and activations for Carnatic Music recordings, extracted from the Saraga Carnatic dataset. The recordings have passed through a Carnatic-aware Analysis/Synthesis framework to convert automatically extracted pitch tracks into ground-truth annotations. This dataset is not meant to be listened to, but to be used as training and evaluation data for the vocal pitch extraction research of Indian Art Music.

The dataset contains a total of 2460 tracks, which generally have a length of 30 seconds, in some cases a bit less. All the tracks have vocals at some point.

The files of this dataset are shared with the following license: Creative Commons Attribution Non Commercial Share Alike 4.0 International

Dataset compiled by: Genís Plaja-Roglans, Thomas Nuttall, Lara Pearson, Xavier Serra, and Marius Miron.

For more information about Saraga Carnatic please refer to https://mtg.github.io/saraga/.

class mirdata.datasets.scms.**Dataset**(*data_home=None*, *version='default'*)

The Saraga-Carnatic-Melody-Synth dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

`download`(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- `ValueError` – if invalid keys are passed to partial_download
- `IOError` – if a downloaded file's checksum is different from expected

`get_mtrack_splits`()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- `AttributeError` – If this dataset does not have multitracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits`(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_track_splits()`

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- **AttributeError** – If this dataset does not have tracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license()`

Print the license

`load_multitracks()`

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
NotImplementedError – If the dataset does not support Multitracks

`load_tracks()`

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns
list – A list of track ids

`track_ids`

Return track ids

Returns
list – A list of track ids

`validate(verbose=True)`

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

`class mirdata.datasets.scms.Track(track_id, data_home, dataset_name, index, metadata)`

Saraga-Carnatic-Melody-Synth Track class

Parameters
track_id (*str*) – track id of the track

Variables

- **artist** (*str*) – artist
- **audio_path** (*str*) – path to the audio file
- **pitch_path** (*str*) – path to the pitch annotation file
- **activations_path** (*str*) – path to the vocal activation annotation file
- **tonic** (*str*) – tonic of the recording
- **gender** (*str*) – gender
- **artist** – instrument of the track
- **title** (*str*) – title
- **train** (*bool*) – indicating if the track belongs to the train or testing set
- **track_id** (*str*) – track id

Other Parameters

- **pitch** (*F0Data*) – vocal pitch time-series
- **activations** (*EventData*) – time regions where the singing voice is present and active

property audio: Tuple[numpy.ndarray, float] | None

The track"s audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track"s data in jams format

Returns
jams.JAMS – the track"s data in jams format

mirdata.datasets.scms.**load_activations**(*fhandle: TextIO*) → *EventData* | None

load a Saraga-Carnatic-Melody-Synth activation annotation file

Parameters
fhandle (*str or file-like*) – str or file-like to note annotation file

Raises
IOError – if file doesn"t exist

Returns
EventData – vocal activations

mirdata.datasets.scms.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a Saraga-Carnatic-Melody-Synth audio file.

Parameters
fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.scms.**load_pitch**(*fhandle: TextIO*) → *F0Data*

load a Saraga-Carnatic-Melody-Synth pitch annotation file

Parameters
fhandle (*str or file-like*) – str or file-like to pitch annotation file

Raises
IOError – if the path doesn"t exist

Returns
F0Data – pitch annotation

2.5.49 slakh

slakh Dataset Loader

Dataset Info

The Synthesized Lakh (Slakh) Dataset is a dataset of multi-track audio and aligned MIDI for music source separation and multi-instrument automatic transcription. Individual MIDI tracks are synthesized from the Lakh MIDI Dataset v0.1 using professional-grade sample-based virtual instruments, and the resulting audio is mixed together to make musical mixtures.

The original release of Slakh, called Slakh2100, contains 2100 automatically mixed tracks and accompanying, aligned MIDI files, synthesized from 187 instrument patches categorized into 34 classes, totaling 145 hours of mixture data.

This loader supports two versions of Slakh: - Slakh2100-redux: a deduplicated version of slakh2100 containing 1710 multitracks - baby-slakh: a mini version with 16k wav audio and only the first 20 tracks

This dataset was created at Mitsubishi Electric Research Labl (MERL) and Interactive Audio Lab at Northwestern University by Ethan Manilow, Gordon Wichern, Prem Seetharaman, and Jonathan Le Roux.

For more information see http://www.slakh.com/

---

class mirdata.datasets.slakh.**Dataset**(*data_home=None*, *version='default'*)

The slakh dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –

- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits`(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_audio`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.slakh.load_audio

`load_midi`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.slakh.load_midi

`load_multif0_from_midi`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.io.load_multif0_from_midi

`load_multitracks`()

Load all multitracks in the dataset

Returns

dict – {*mtrack_id*: multitrack data}

Raises

`NotImplementedError` – If the dataset does not support Multitracks

load_notes_from_midi(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.io.load_notes_from_midi

load_tracks()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

mtrack_ids

Return track ids

Returns
list – A list of track ids

track_ids

Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

mirdata.datasets.slakh.**MIXING_GROUPS = {'bass': [32, 33, 34, 35, 36, 37, 38, 39], 'drums': [128], 'guitar': [24, 25, 26, 27, 28, 29, 30, 31], 'piano': [0, 1, 2, 3, 4, 5, 6, 7]}**

Mixing group to program number mapping

class mirdata.datasets.slakh.**MultiTrack**(*mtrack_id*, *data_home*, *dataset_name*, *index*, *track_class*, *metadata*)

slakh multitrack class, containing information about the mix and the set of associated stems

Variables

- **mtrack_id** (*str*) – track id
- **tracks** (*dict*) – {track_id: Track}
- **track_audio_property** (*str*) – the name of the attribute of Track which returns the audio to be mixed
- **mix_path** (*str*) – path to the multitrack mix audio
- **midi_path** (*str*) – path to the full midi data used to generate the mixture
- **metadata_path** (*str*) – path to the multitrack metadata file
- **split** (*str or None*) – one of 'train', 'validation', 'test', or 'omitted'. 'omitted' tracks are part of slakh2100-redux which were found to be duplicates in the original slakh2011.
- **data_split** (*str or None*) – equivalent to split (deprecated in 0.3.6)

- **uuid** (*str*) – File name of the original MIDI file from Lakh, sans extension
- **lakh_midi_dir** (*str*) – Path to the original MIDI file from a fresh download of Lakh
- **normalized** (*bool*) – whether the mix and stems were normalized according to the ITU-R BS.1770-4 spec
- **overall_gain** (*float*) – gain applied to every stem to make sure mixture does not clip when stems are summed

Other Parameters

- **midi** (*PrettyMIDI*) – midi data used to generate the mixture audio
- **notes** (*NoteData*) – note representation of the midi data
- **multif0** (*MultiF0Data*) – multif0 representation of the midi data

property audio: Tuple[numpy.ndarray, float] | None

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_mix()

Create a linear mixture given a subset of tracks.

Parameters
track_keys (*list*) – list of track keys to mix together

Returns
np.ndarray – mixture audio with shape (n_samples, n_channels)

get_path(*key*)

Get absolute path to multitrack audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

get_random_target(*n_tracks=None*, *min_weight=0.3*, *max_weight=1.0*)

Get a random target by combining a random selection of tracks with random weights

Parameters

- **n_tracks** (*int or None*) – number of tracks to randomly mix. If None, uses all tracks
- **min_weight** (*float*) – minimum possible weight when mixing
- **max_weight** (*float*) – maximum possible weight when mixing

Returns

- np.ndarray - mixture audio with shape (n_samples, n_channels)
- list - list of keys of included tracks
- list - list of weights used to mix tracks

`get_submix_by_group`(*target_groups*)

Create submixes grouped by instrument type. Creates one submix per target group, plus one additional "other" group for any remaining sources. Only tracks with available audio are mixed.

Parameters

target_groups (*list*) – List of target groups. Elements should be one of MIXING_GROUPS, e.g. ["bass", "guitar"]

Returns

** submixes (dict)* – {group: audio_signal} of submixes * groups (dict): {group: list of track ids} of submixes

`get_target`(*track_keys*, *weights=None*, *average=True*, *enforce_length=True*)

Get target which is a linear mixture of tracks

Parameters

- **track_keys** (*list*) – list of track keys to mix together
- **weights** (*list or None*) – list of positive scalars to be used in the average
- **average** (*bool*) – if True, computes a weighted average of the tracks if False, computes a weighted sum of the tracks
- **enforce_length** (*bool*) – If True, raises ValueError if the tracks are not the same length. If False, pads audio with zeros to match the length of the longest track

Returns

np.ndarray – target audio with shape (n_channels, n_samples)

Raises

`ValueError` – if sample rates of the tracks are not equal if enforce_length=True and lengths are not equal

`to_jams`()

Jams: the track's data in jams format

`class mirdata.datasets.slakh.Track`(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

slakh Track class, for individual stems

Variables

- `audio_path` (*str or None*) – path to the track's audio file. For some unusual tracks, such as sound effects, there is no audio and this attribute is None.
- `split` (*str or None*) – one of 'train', 'validation', 'test', or 'omitted'. 'omitted' tracks are part of slakh2100-redux which were found to be duplicates in the original slakh2011. In baby slakh there are no splits, so this attribute is None.
- `data_split` (*str or None*) – equivalent to split (deprecated in 0.3.6)
- `metadata_path` (*str*) – path to the multitrack's metadata file
- `midi_path` (*str or None*) – path to the track's midi file. For some unusual tracks, such as sound effects, there is no midi and this attribute is None.
- `mtrack_id` (*str*) – the track's multitrack id
- `track_id` (*str*) – track id
- `instrument` (*str*) – MIDI instrument class, see link for details: https://en.wikipedia.org/wiki/General_MIDI#Program_change_events

- **integrated_loudness** (*float*) – integrated loudness (dB) of this track as calculated by the ITU-R BS.1770-4 spec
- **is_drum** (*bool*) – whether the "drum" flag is true for this MIDI track
- **midi_program_name** (*str*) – MIDI instrument program name
- **plugin_name** (*str*) – patch/plugin name that rendered the audio file
- **mixing_group** (*str*) – which mixing group the track belongs to. One of MIXING_GROUPS.
- **program_number** (*int*) – MIDI instrument program number

Other Parameters

- **midi** (*PrettyMIDI*) – midi data used to generate the audio
- **notes** (*NoteData or None*) – note representation of the midi data. If there are no notes in the midi file, returns None.
- **multif0** (*MultiF0Data or None*) – multif0 representaation of the midi data. If there are no notes in the midi file, returns None.

property audio: Tuple[numpy.ndarray, float] | None

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Jams: the track's data in jams format

mirdata.datasets.slakh.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a slakh audio file.

Parameters
fhandle (*str or file-like*) – path or file-like object pointing to an audio file

Returns

- np.ndarray - the audio signal
- float - The sample rate of the audio file

2.5.50 tinysol

TinySOL Dataset Loader.

Dataset Info

TinySOL is a dataset of 2913 samples, each containing a single musical note from one of 14 different instruments:

- Bass Tuba
- French Horn
- Trombone
- Trumpet in C
- Accordion
- Contrabass
- Violin
- Viola
- Violoncello
- Bassoon
- Clarinet in B-flat
- Flute
- Oboe
- Alto Saxophone

These sounds were originally recorded at Ircam in Paris (France) between 1996 and 1999, as part of a larger project named Studio On Line (SOL). Although SOL contains many combinations of mutes and extended playing techniques, TinySOL purely consists of sounds played in the so-called "ordinary" style, and in absence of mute.

TinySOL can be used for education and research purposes. In particular, it can be employed as a dataset for training and/or evaluating music information retrieval (MIR) systems, for tasks such as instrument recognition or fundamental frequency estimation. For this purpose, we provide an official 5-fold split of TinySOL as a metadata attribute. This split has been carefully balanced in terms of instrumentation, pitch range, and dynamics. For the sake of research reproducibility, we encourage users of TinySOL to adopt this split and report their results in terms of average performance across folds.

We encourage TinySOL users to subscribe to the Ircam Forum so that they can have access to larger versions of SOL.

For more details, please visit: https://www.orch-idea.org/

---

class mirdata.datasets.tinysol.**Dataset**(*data_home=None*, *version='default'*)

The tinysol dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –

- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits`(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_audio`(**args*, ***kwargs*)

Deprecated since version 0.3.4: Use mirdata.datasets.tinysol.load_audio

`load_multitracks`()

Load all multitracks in the dataset

Returns

dict – {*mtrack_id*: multitrack data}

Raises

`NotImplementedError` – If the dataset does not support Multitracks

`load_tracks`()

Load all tracks in the dataset

Returns

dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

mtrack_ids

Return track ids

Returns
list – A list of track ids

track_ids

Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.tinysol.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

tinysol Track class

Parameters
track_id (*str*) – track id of the track

Variables

- **audio_path** (*str*) – path of the audio file
- **dynamics** (*str*) – dynamics abbreviation. Ex: pp, mf, ff, etc.
- **dynamics_id** (*int*) – pp=0, p=1, mf=2, f=3, ff=4
- **family** (*str*) – instrument family encoded by its English name
- **instance_id** (*int*) – instance ID. Either equal to 0, 1, 2, or 3.
- **instrument_abbr** (*str*) – instrument abbreviation
- **instrument_full** (*str*) – instrument encoded by its English name
- **is_resampled** (*bool*) – True if this sample was pitch-shifted from a neighbor; False if it was genuinely recorded.
- **pitch** (*str*) – string containing English pitch class and octave number
- **pitch_id** (*int*) – MIDI note index, where middle C ("C4") corresponds to 60
- **string_id** (*NoneType*) – string ID. By musical convention, the first string is the highest. On wind instruments, this is replaced by *None*.
- **technique_abbr** (*str*) – playing technique abbreviation
- **technique_full** (*str*) – playing technique encoded by its English name
- **track_id** (*str*) – track id

property audio: Tuple[numpy.ndarray, float] | None

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.tinysol.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a TinySOL audio file.

Parameters
fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

2.5.51 tonality_classicaldb

Tonality classicalDB Dataset Loader

Dataset Info

The Tonality classicalDB Dataset includes 881 classical musical pieces across different styles from s.XVII to s.XX annotated with single-key labels.

Tonality classicalDB Dataset was created as part of:

```
Gómez, E. (2006). PhD Thesis. Tonal description of music audio signals.
Department of Information and Communication Technologies.
```

This dataset is mainly intended to assess the performance of computational key estimation algorithms in classical music.

2020 note: The audio is privates. If you don't have the original audio collection, you could create it from your private collection because most of the recordings are well known. To this end, we provide musicbrainz metadata. Moreover, we have added the spectrum and HPCP chromagram of each audio.

This dataset can be used with mirdata library: https://github.com/mir-dataset-loaders/mirdata

Spectrum features have been computed as is shown here: https://github.com/mir-dataset-loaders/mirdata-notebooks/blob/master/Tonality_classicalDB/ClassicalDB_spectrum_features.ipynb

HPCP chromagram has been computed as is shown here: https://github.com/mir-dataset-loaders/mirdata-notebooks/blob/master/Tonality_classicalDB/ClassicalDB_HPCP_features.ipynb

Musicbrainz metadata has been computed as is shown here: https://github.com/mir-dataset-loaders/mirdata-notebooks/blob/master/Tonality_classicalDB/ClassicalDB_musicbrainz_metadata.ipynb

---

class mirdata.datasets.tonality_classicaldb.**Dataset**(*data_home=None*, *version='default'*)

The tonality_classicaldb dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns

str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.

- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

`get_mtrack_splits()`

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of mtrack_ids

`get_random_mtrack_splits(splits, seed=42, split_names=None)`

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_random_track_splits(splits, seed=42, split_names=None)`

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

`get_track_splits()`

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- **AttributeError** – If this dataset does not have tracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of track_ids

`license()`

Print the license

`load_audio(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.tonality_classicaldb.load_audio

`load_hpcp(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.tonality_classicaldb.load_hpcp

`load_key(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.tonality_classicaldb.load_key

`load_multitracks()`

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

`load_musicbrainz(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.tonality_classicaldb.load_musicbrainz

`load_spectrum(*args, **kwargs)`

Deprecated since version 0.3.4: Use mirdata.datasets.tonality_classicaldb.load_spectrum

`load_tracks()`

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
`NotImplementedError` – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns
list – A list of track ids

`track_ids`

Return track ids

Returns
list – A list of track ids

`validate(verbose=True)`

Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.tonality_classicaldb.**Track**(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

tonality_classicaldb track class

Parameters
track_id (*str*) – track id of the track

Variables

- **audio_path** (*str*) – track audio path
- **key_path** (*str*) – key annotation path
- **title** (*str*) – title of the track
- **track_id** (*str*) – track id

Other Parameters

- **key** (*str*) – key annotation
- **spectrum** (*np.array*) – computed audio spectrum
- **hpcp** (*np.array*) – computed hpcp
- **musicbrainz_metadata** (*dict*) – MusicBrainz metadata

property audio: Tuple[numpy.ndarray, float] | None

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.tonality_classicaldb.**load_audio**(*fhandle: BinaryIO*) → Tuple[numpy.ndarray, float]

Load a Tonality classicalDB audio file.

Parameters
fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.tonality_classicaldb.**load_hpcp**(*fhandle: TextIO*) → numpy.ndarray

Load Tonality classicalDB HPCP feature from a file

Parameters
fhandle (*str or file-like*) – File-like object or path to HPCP file

Returns
np.ndarray – loaded HPCP data

mirdata.datasets.tonality_classicaldb.**load_key**(*fhandle: TextIO*) → str

Load Tonality classicalDB format key data from a file

Parameters
fhandle (*str or file-like*) – File-like object or path to key annotation file

Returns
str – musical key data

mirdata.datasets.tonality_classicaldb.**load_musicbrainz**(*fhandle: TextIO*) → Dict[Any, Any]

Load Tonality classicalDB musicbraiz metadata from a file

Parameters
fhandle (*str or file-like*) – File-like object or path to musicbrainz metadata file

Returns
dict – musicbrainz metadata

mirdata.datasets.tonality_classicaldb.**load_spectrum**(*fhandle: TextIO*) → numpy.ndarray

Load Tonality classicalDB spectrum data from a file

Parameters
fhandle (*str or file-like*) – File-like object or path to spectrum file

Returns
np.ndarray – spectrum data

2.5.52 tonas

TONAS Loader

Dataset Info

This dataset contains a music collection of 72 sung excerpts representative of three a cappella singing styles (Deblas, and two variants of Martinete). It has been developed within the COFLA research project context. The distribution is as follows: 1. 16 Deblas 2. 36 Martinete 1 3. 20 Martinete 2

This collection was built in the context of a study on similarity and style classification of flamenco a cappella singing styles (Tonas) by the flamenco expert Dr. Joaquin Mora, Universidad de Sevilla.

We refer to (Mora et al. 2010) for a comprehensive description of the considered styles and their musical characteristics. All 72 excerpts are monophonic, their average duration is 30 seconds and there is enough variability for a proper evaluation of our methods, including a variety of singers, recording conditions, presence of percussion, clapping, background voices and noise. We also provide manual melodic transcriptions, generated by the COFLA team and Cristina López Gómez.

The annotations are represented by specifying the value (in this case, Notes and F0) at the related timestamps. TONAS' note and F0 annotations also have "Energy" information, which refers to the average energy value through all the frames in which a note or a F0 value is comprised.

Using this dataset: TONAS dataset can be obtained upon request. Please refer to this link: https://zenodo.org/record/1290722 to request access and follow the indications of the .download() method for a proper storing and organization of the TONAS dataset.

Citing this dataset: When TONAS is used for academic research, we would highly appreciate if scientific publications of works partly based on the TONAS dataset quote the following publication: - Music material: Mora, J., Gomez, F., Gomez, E., Escobar-Borrego, F.J., Diaz-Banez, J.M. (2010). Melodic Characterization and Similarity in A Cappella Flamenco Cantes. 11th International Society for Music Information Retrieval Conference (ISMIR 2010). - Transcriptions: Gomez, E., Bonada, J. (in Press). Towards Computer-Assisted Flamenco Transcription: An Experimental Comparison of Automatic Transcription Algorithms As Applied to A Cappella Singing. Computer Music Journal.

---

class mirdata.datasets.tonas.**Dataset**(*data_home=None*, *version='default'*)

The TONAS dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns

str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded

- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns

dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

get_random_track_splits(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns

dict – a dictionary containing the elements in each split

get_track_splits()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- **AttributeError** – If this dataset does not have tracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license()`
Print the license

`load_audio(*args, **kwargs)`
Deprecated since version 0.3.4: Use mirdata.datasets.tonas.load_audio

`load_f0(*args, **kwargs)`
Deprecated since version 0.3.4: Use mirdata.datasets.tonas.load_f0

`load_multitracks()`
Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
NotImplementedError – If the dataset does not support Multitracks

`load_notes(*args, **kwargs)`
Deprecated since version 0.3.4: Use mirdata.datasets.tonas.load_notes

`load_tracks()`
Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
NotImplementedError – If the dataset does not support Tracks

`mtrack_ids`
Return track ids

Returns
list – A list of track ids

`track_ids`
Return track ids

Returns
list – A list of track ids

`validate(verbose=True)`
Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

`class mirdata.datasets.tonas.Track(track_id, data_home, dataset_name, index, metadata)`
TONAS track class

Parameters

- **track_id** (*str*) – track id of the track

- **data_home** (*str*) – Local path where the dataset is stored. If *None*, looks for the data in the default directory, *~/mir_datasets/TONAS*

Variables

- **f0_path** (*str*) – local path where f0 melody annotation file is stored
- **notes_path** (*str*) – local path where notation annotation file is stored
- **audio_path** (*str*) – local path where audio file is stored
- **track_id** (*str*) – track id
- **singer** (*str*) – performing singer (cantaor)
- **title** (*str*) – title of the track song
- **tuning_frequency** (*float*) – tuning frequency of the symbolic notation

Other Parameters

- **f0_automatic** (*F0Data*) – automatically extracted f0
- **f0_corrected** (*F0Data*) – manually corrected f0 annotations
- **notes** (*NoteData*) – annotated notes

property audio: Tuple[numpy.ndarray, float]

The track's audio

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

to_jams()

Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

mirdata.datasets.tonas.**load_audio**(*fhandle: str*) → Tuple[numpy.ndarray, float]

Load a TONAS audio file.

Parameters
fhandle (*str*) – path to an audio file

Returns

- np.ndarray - the mono audio signal
- float - The sample rate of the audio file

mirdata.datasets.tonas.**load_f0**(*fpath: str*, *corrected: bool*) → *F0Data* | None

Load TONAS f0 annotations

Parameters

- **fpath** (*str*) – path pointing to f0 annotation file
- **corrected** (*bool*) – if True, loads manually corrected frequency values otherwise, loads automatically extracted frequency values

Returns

F0Data – predominant f0 melody

mirdata.datasets.tonas.**load_notes**(*fhandle: TextIO*) → *NoteData* | None

Load TONAS note data from the annotation files

Parameters

fhandle (*str or file-like*) – path or file-like object pointing to a notes annotation file

Returns

NoteData – note annotations

2.5.53 vocadito

vocadito Dataset Loader

Dataset Info

vocadito is a dataset of 40 short excerpts of solo, monophonic singing. The excerpts are sung in 7 different languages by singers with varying of levels of training, and are recorded on a variety of devices.

Annotations are labeled by trained musicians. For each excerpt, we provide:

frame-level f0 annotations 2 versions of note annotations (from 2 different annotators) lyrics language

For more details, please visit: https://zenodo.org/record/5578807

---

class mirdata.datasets.vocadito.**Dataset**(*data_home=None*, *version='default'*)

The vocadito dataset

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

choice_multitrack()

Choose a random multitrack

Returns
Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns
Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list

- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_random_track_splits`(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_track_splits`()

Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license`()

Print the license

`load_multitracks`()

Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

`load_tracks`()

Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
`NotImplementedError` – If the dataset does not support Tracks

`mtrack_ids`

Return track ids

Returns
list – A list of track ids

track_ids
Return track ids

Returns
list – A list of track ids

validate(*verbose=True*)
Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.datasets.vocadito.Track(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)
vocadito Track class

Parameters
track_id (*str*) – track id of the track

Variables

- **audio_path** (*str*) – path to the track's audio file
- **f0_path** (*str*) – path to the track's f0 annotation file
- **lyrics_path** (*str*) – path to the track's lyric annotation file
- **notes_a1_path** (*str*) – path to the track's note annotation file for annotator A1
- **notes_a2_path** (*str*) – path to the track's note annotation file for annotator A2
- **track_id** (*str*) – track id
- **singer_id** (*str*) – singer id
- **average_pitch_midi** (*int*) – Average pitch in midi, computed from the f0 annotation
- **language** (*str*) – The track's language. May contain multiple languages.

Other Parameters

- **f0** (*F0Data*) – human-annotated singing voice pitch
- **lyrics** (*List[List[str]]*) – human-annotated lyrics
- **notes_a1** (*NoteData*) – human-annotated notes by annotator A1
- **notes_a2** (*NoteData*) – human-annotated notes by annotator A2

property audio: Tuple[numpy.ndarray, float] | None
solo vocal audio (mono)

Returns

- np.ndarray - audio signal
- float - sample rate

get_path(*key*)
Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

`to_jams()`
Get the track's data in jams format

Returns
jams.JAMS – the track's data in jams format

`mirdata.datasets.vocadito.load_audio(fhandle: BinaryIO) → Tuple[numpy.ndarray, float]`
Load vocadito vocal audio

Parameters
fhandle (*str or file-like*) – File-like object or path to audio file

Returns

- np.ndarray - audio signal
- float - sample rate

`mirdata.datasets.vocadito.load_f0(fhandle: TextIO) → F0Data`
Load a vocadito f0 annotation

Parameters
fhandle (*str or file-like*) – File-like object or path to f0 annotation file

Raises
IOError – If f0_path does not exist

Returns
F0Data – the f0 annotation data

`mirdata.datasets.vocadito.load_lyrics(fhandle: TextIO) → List[List[str]]`
Load a lyrics annotation

Parameters
fhandle (*str or file-like*) – File-like object or path to lyric annotation file

Raises
IOError – if lyrics_path does not exist

Returns
LyricData – lyric annotation data

`mirdata.datasets.vocadito.load_notes(fhandle: TextIO) → NoteData | None`
load a note annotation file

Parameters
fhandle (*str or file-like*) – str or file-like to note annotation file

Raises
IOError – if file doesn't exist

Returns
NoteData – note annotation

2.6 Core

Core mirdata classes

class mirdata.core.**Dataset**(*data_home=None*, *version='default'*, *name=None*, *track_class=None*, *multitrack_class=None*, *bibtex=None*, *indexes=None*, *remotes=None*, *download_info=None*, *license_info=None*)

mirdata Dataset class

Variables

- **data_home** (*str*) – path where mirdata will look for the dataset
- **version** (*str*) –
- **name** (*str*) – the identifier of the dataset
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **indexes** (*dict or None*) –
- **remotes** (*dict or None*) – data to be downloaded
- **readme** (*str*) – information about the dataset
- **track** (*function*) – a function mapping a track_id to a mirdata.core.Track
- **multitrack** (*function*) – a function mapping a mtrack_id to a mirdata.core.Multitrack

__init__(*data_home=None*, *version='default'*, *name=None*, *track_class=None*, *multitrack_class=None*, *bibtex=None*, *indexes=None*, *remotes=None*, *download_info=None*, *license_info=None*)

Dataset init method

Parameters

- **data_home** (*str or None*) – path where mirdata will look for the dataset
- **name** (*str or None*) – the identifier of the dataset
- **track_class** (*mirdata.core.Track or None*) – a Track class
- **multitrack_class** (*mirdata.core.Multitrack or None*) – a Multitrack class
- **bibtex** (*str or None*) – dataset citation/s in bibtex format
- **remotes** (*dict or None*) – data to be downloaded
- **download_info** (*str or None*) – download instructions or caveats
- **license_info** (*str or None*) – license of the dataset

choice_multitrack()

Choose a random multitrack

Returns

Multitrack – a Multitrack object instantiated by a random mtrack_id

choice_track()

Choose a random track

Returns

Track – a Track object instantiated by a random track_id

cite()

Print the reference

property default_path

Get the default path for the dataset

Returns
str – Local path to the dataset

download(*partial_download=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally print a message.

Parameters

- **partial_download** (*list or None*) – A list of keys of remotes to partially download. If None, all data is downloaded
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete any zip/tar files after extracting.
- **allow_invalid_checksum** (*bool*) – Allow invalid checksums of the downloaded data. Useful sometimes behind some proxies that inspection the downloaded data. When having a different checksum promts a warn instead of raising an exception

Raises

- **ValueError** – if invalid keys are passed to partial_download
- **IOError** – if a downloaded file's checksum is different from expected

get_mtrack_splits()

Get predetermined multitrack splits (e.g. train/ test) released alongside this dataset.

Raises

- **AttributeError** – If this dataset does not have multitracks
- **NotImplementedError** – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of mtrack_ids

get_random_mtrack_splits(*splits*, *seed=42*, *split_names=None*)

Split the multitracks into partitions, e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list
- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

get_random_track_splits(*splits*, *seed=42*, *split_names=None*)

Split the tracks into partitions e.g. training, validation, test

Parameters

- **splits** (*list of float*) – a list of floats that should sum up 1. It will return as many splits as elements in the list

- **seed** (*int*) – the seed used for the random generator, in order to enhance reproducibility. Defaults to 42
- **split_names** (*list*) – list of keys to use in the output dictionary

Returns
dict – a dictionary containing the elements in each split

`get_track_splits()`
Get predetermined track splits (e.g. train/ test) released alongside this dataset

Raises

- `AttributeError` – If this dataset does not have tracks
- `NotImplementedError` – If this dataset does not have predetermined splits

Returns
dict – splits, keyed by split name and with values of lists of track_ids

`license()`
Print the license

`load_multitracks()`
Load all multitracks in the dataset

Returns
dict – {*mtrack_id*: multitrack data}

Raises
`NotImplementedError` – If the dataset does not support Multitracks

`load_tracks()`
Load all tracks in the dataset

Returns
dict – {*track_id*: track data}

Raises
`NotImplementedError` – If the dataset does not support Tracks

`mtrack_ids`
Return track ids

Returns
list – A list of track ids

`track_ids`
Return track ids

Returns
list – A list of track ids

`validate(verbose=True)`
Validate if the stored dataset is a valid version

Parameters
verbose (*bool*) – If False, don't print output

Returns

- list - files in the index but are missing locally
- list - files which have an invalid checksum

class mirdata.core.**Index**(*filename: str*, *url: str | None = None*, *checksum: str | None = None*, *partial_download: List[str] | None = None*)

Class for storing information about dataset indexes.

Parameters

- **filename** (*str*) – The index filename (not path), e.g. "example_dataset_index_1.2.json"
- **url** (*str or None*) – None if index is not remote, or a url to download from
- **checksum** (*str or None*) – None if index is not remote, or the md5 checksum of the file
- **partial_download** (*list or None*) – if provided, specifies a subset of Dataset.remotes corresponding to this index to be downloaded. If None, all Dataset.remotes will be downloaded when calling Dataset.download()

Variables

- **remote** (download_utils.RemoteFileMetadata *or None*) – None if index is not remote, or a RemoteFileMetadata object
- **partial_download** (*list or None*) – a list of keys to partially download, or None

get_path(*data_home: str*) → str

Get the absolute path to the index file

Parameters
data_home (*str*) – Path where the dataset's data lives

Returns
str – absolute path to the index file

class mirdata.core.**MultiTrack**(*mtrack_id*, *data_home*, *dataset_name*, *index*, *track_class*, *metadata*)

MultiTrack class.

A multitrack class is a collection of track objects and their associated audio that can be mixed together. A multitrack is itself a Track, and can have its own associated audio (such as a mastered mix), its own metadata and its own annotations.

__init__(*mtrack_id*, *data_home*, *dataset_name*, *index*, *track_class*, *metadata*)

Multitrack init method. Sets boilerplate attributes, including:

- mtrack_id
- _dataset_name
- _data_home
- _multitrack_paths
- _multitrack_metadata

Parameters

- **mtrack_id** (*str*) – multitrack id
- **data_home** (*str*) – path where mirdata will look for the dataset
- **dataset_name** (*str*) – the identifier of the dataset
- **index** (*dict*) – the dataset's file index
- **metadata** (*function or None*) – a function returning a dictionary of metadata or None

`get_mix()`

Create a linear mixture given a subset of tracks.

Parameters
track_keys (*list*) – list of track keys to mix together

Returns
np.ndarray – mixture audio with shape (n_samples, n_channels)

`get_path(key)`

Get absolute path to multitrack audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

`get_random_target(n_tracks=None, min_weight=0.3, max_weight=1.0)`

Get a random target by combining a random selection of tracks with random weights

Parameters

- **n_tracks** (*int or None*) – number of tracks to randomly mix. If None, uses all tracks
- **min_weight** (*float*) – minimum possible weight when mixing
- **max_weight** (*float*) – maximum possible weight when mixing

Returns

- np.ndarray - mixture audio with shape (n_samples, n_channels)
- list - list of keys of included tracks
- list - list of weights used to mix tracks

`get_target(track_keys, weights=None, average=True, enforce_length=True)`

Get target which is a linear mixture of tracks

Parameters

- **track_keys** (*list*) – list of track keys to mix together
- **weights** (*list or None*) – list of positive scalars to be used in the average
- **average** (*bool*) – if True, computes a weighted average of the tracks if False, computes a weighted sum of the tracks
- **enforce_length** (*bool*) – If True, raises ValueError if the tracks are not the same length. If False, pads audio with zeros to match the length of the longest track

Returns
np.ndarray – target audio with shape (n_channels, n_samples)

Raises
`ValueError` – if sample rates of the tracks are not equal if enforce_length=True and lengths are not equal

`class mirdata.core.Track(track_id, data_home, dataset_name, index, metadata)`

Track base class

See the docs for each dataset loader's Track class for details

__init__(*track_id*, *data_home*, *dataset_name*, *index*, *metadata*)

Track init method. Sets boilerplate attributes, including:

- `track_id`
- `_dataset_name`
- `_data_home`
- `_track_paths`
- `_track_metadata`

Parameters

- **track_id** (*str*) – track id
- **data_home** (*str*) – path where mirdata will look for the dataset
- **dataset_name** (*str*) – the identifier of the dataset
- **index** (*dict*) – the dataset's file index
- **metadata** (*function or None*) – a function returning a dictionary of metadata or None

get_path(*key*)

Get absolute path to track audio and annotations. Returns None if the path in the index is None

Parameters
key (*string*) – Index key of the audio or annotation type

Returns
str or None – joined path string or None

class mirdata.core.**cached_property**(*func*)

Cached propery decorator

A property that is only computed once per instance and then replaces itself with an ordinary attribute. Deleting the attribute resets the property. Source: https://github.com/bottlepy/bottle/commit/fa7733e075da0d790d809aa3d2f53071897e6f76

mirdata.core.**docstring_inherit**(*parent*)

Decorator function to inherit docstrings from the parent class.

Adds documented Attributes from the parent to the child docs.

2.7 Annotations

mirdata annotation data types

mirdata.annotations.**AMPLITUDE_UNITS = {'binary': '0 or 1', 'energy': 'energy value, measured as the sum of a squared signal', 'likelihood': 'score between 0 and 1', 'velocity': 'MIDI velocity between 0 and 127'}**

Amplitude/voicing units

class mirdata.annotations.**Annotation**

Annotation base class

mirdata.annotations.**BEAT_POSITION_UNITS = {'bar_fraction': 'beat position as fractions of bars, e.g. 0.25', 'bar_index': 'beat index within a bar, 1-indexed', 'global_fraction': 'bar_frac, but where the integer part indicates the bar. e.g. 4.25', 'global_index': 'beat index within full track, 1-indexed'}**

Beat position units

class mirdata.annotations.**BeatData**(*times*, *time_unit*, *positions*, *position_unit*, *confidence=None*, *confidence_unit=None*)

BeatData class

Variables

- **times** (*np.ndarray*) – array of time stamps with positive, strictly increasing values
- **time_unit** (*str*) – time unit, one of TIME_UNITS
- **positions** (*np.ndarray*) – array of beat positions in the format of position_unit. For all units, values of 0 indicate beats which fall outside of a measure.
- **position_unit** (*str*) – beat position unit, one of BEAT_POSITION_UNITS
- **confidence** (*np.ndarray*) – array of confidence values
- **confidence_unit** (*str*) – confidence unit, one of AMPLITUDE_UNITS

mirdata.annotations.**CHORD_UNITS = {'harte': 'chords in harte format, e.g. Ab:maj7', 'jams': "chords in jams 'chord' format", 'open': 'no strict schema or units'}**

Chord units

class mirdata.annotations.**ChordData**(*intervals*, *interval_unit*, *labels*, *label_unit*, *confidence=None*, *confidence_unit=None*)

ChordData class

Variables

- **intervals** (*np.ndarray*) – (n x 2) array of intervals in the form [start_time, end_time]. Times should be positive and intervals should have non-negative duration
- **interval_unit** (*str*) – unit of the time values in intervals. One of TIME_UNITS.
- **labels** (*list*) – list chord labels (as strings)
- **label_unit** (*str*) – chord label schema
- **confidence** (*np.ndarray or None*) – array of confidence values
- **confidence_unit** (*str or None*) – confidence unit, one of AMPLITUDE_UNITS

mirdata.annotations.**EVENT_UNITS = {'open': 'no scrict schema or units'}**

Event units

class mirdata.annotations.**EventData**(*intervals*, *interval_unit*, *events*, *event_unit*)

EventData class

Variables

- **intervals** (*np.ndarray*) – (n x 2) array of intervals in the form [start_time, end_time]. Times should be positive and intervals should have non-negative duration
- **interval_unit** (*str*) – unit of the time values in intervals. One of TIME_UNITS.
- **interval_unit** – interval units, one of TIME_UNITS
- **events** (*list*) – list of event labels (as strings)

- **event_unit** (*str*) – event units, one of EVENT_UNITS

class mirdata.annotations.**F0Data**(*times*, *time_unit*, *frequencies*, *frequency_unit*, *voicing*, *voicing_unit*, *confidence=None*, *confidence_unit=None*)

F0Data class

Variables

- **times** (*np.ndarray*) – array of time stamps (as floats) with positive, strictly increasing values
- **time_unit** (*str*) – time unit, one of TIME_UNITS
- **frequencies** (*np.ndarray*) – array of frequency values (as floats)
- **frequency_unit** (*str*) – frequency unit, one of PITCH_UNITS
- **voicing** (*np.ndarray*) – array of voicing values, indicating whether or not a time frame has an active pitch
- **voicing_unit** (*str*) – voicing unit, one of VOICING_UNITS
- **confidence** (*np.ndarray or None*) – array of confidence values
- **confidence_unit** (*str or None*) – confidence unit, one of AMPLITUDE_UNITS

resample(*times_new*, *times_new_unit*)

Resample the annotation to a new time scale. This function is adapted from: https://github.com/craffel/mir_eval/blob/master/mir_eval/melody.py#L212

Parameters

- **times_new** (*np.ndarray*) – new time base, in units of times_new_unit
- **times_new_unit** (*str*) – time unit, one of TIME_UNITS

Returns

F0Data – F0 data sampled at new time scale

to_matrix(*time_scale*, *time_scale_unit*, *frequency_scale*, *frequency_scale_unit*, *amplitude_unit='binary'*)

Convert f0 data to a matrix (piano roll) defined by a time and frequency scale

Parameters

- **time_scale** (*np.array*) – times in units time_unit
- **time_scale_unit** (*str*) – time scale units, one of TIME_UNITS
- **frequency_scale** (*np.array*) – frequencies in frequency_unit
- **frequency_scale_unit** (*str*) – frequency scale units, one of PITCH_UNITS
- **amplitude_unit** (*str*) – amplitude units, one of AMPLITUDE_UNITS Defaults to "binary".

Returns

np.ndarray – 2D matrix of shape len(time_scale) x len(frequency_scale)

to_mir_eval()

Convert units and format to what is expected by mir_eval.melody.evaluate

Returns

- times (np.ndarray) - uniformly spaced times in seconds
- frequencies (np.ndarray) - frequency values in hz

- voicing (np.ndarray) - voicings, as likelihood values

to_multif0()

Convert annotation to multif0 format

Returns

MultiF0Data – data in multif0 format

to_sparse_index(*time_scale*, *time_scale_unit*, *frequency_scale*, *frequency_scale_unit*, *amplitude_unit='binary'*)

Convert F0 annotation to sparse matrix indices for a time-frequency matrix.

Parameters

- **time_scale** (*np.array*) – times in units time_unit
- **time_scale_unit** (*str*) – time scale units, one of TIME_UNITS
- **frequency_scale** (*np.array*) – frequencies in frequency_unit
- **frequency_scale_unit** (*str*) – frequency scale units, one of PITCH_UNITS
- **amplitude_unit** (*str*) – amplitude units, one of AMPLITUDE_UNITS Defaults to "binary".

Returns

** sparse_index (np.ndarray)* – Array of sparce indices [(time_index, frequency_index)] * amplitude (np.ndarray): Array of amplitude values for each index

mirdata.annotations.**KEY_UNITS = {'key_mode': 'key labels in key-mode format, e.g. G#:minor'}**

Key units

class mirdata.annotations.**KeyData**(*intervals*, *interval_unit*, *keys*, *key_unit*)

KeyData class

Variables

- **intervals** (*np.ndarray*) – (n x 2) array of intervals in the form [start_time, end_time]. Times should be positive and intervals should have non-negative duration
- **interval_unit** (*str*) – unit of the time values in intervals. One of TIME_UNITS.
- **keys** (*list*) – list key labels (as strings)
- **key_unit** (*str*) – key unit, one of KEY_UNITS

mirdata.annotations.**LYRIC_UNITS = {'pronunciations_open': 'lyric pronunciations, no strict schema', 'syllable_open': 'lyrics segmented by syllable, no strict schema', 'words': 'lyrics as words or phrases'}**

Lyric units

class mirdata.annotations.**LyricData**(*intervals*, *interval_unit*, *lyrics*, *lyric_unit*)

LyricData class

Variables

- **intervals** (*np.ndarray*) – (n x 2) array of intervals in the form [start_time, end_time]. Times should be positive and intervals should have non-negative duration
- **interval_unit** (*str*) – unit of the time values in intervals. One of TIME_UNITS.
- **lyrics** (*list*) – list of lyrics (as strings)

- **lyric_unit** (*str*) – lyric unit, one of LYRIC_UNITS

class mirdata.annotations.**MultiF0Data**(*times*, *time_unit*, *frequency_list*, *frequency_unit*, *confidence_list=None*, *confidence_unit=None*)

MultiF0Data class

Variables

- **times** (*np.ndarray*) – array of time stamps (as floats) with positive, strictly increasing values
- **time_unit** (*str*) – time unit, one of TIME_UNITS
- **frequency_list** (*list*) – list of lists of frequency values (as floats)
- **frequency_unit** (*str*) – frequency unit, one of PITCH_UNITS
- **confidence_list** (*np.ndarray or None*) – list of lists of confidence values
- **confidence_unit** (*str or None*) – confidence unit, one of AMPLITUDE_UNITS

resample(*times_new*, *times_new_unit*)

Resample annotation to a new time scale. This function is adapted from: https://github.com/craffel/mir_eval/blob/master/mir_eval/multipitch.py#L104

Parameters

- **times_new** (*np.array*) – array of new time scale values
- **times_new_unit** (*str*) – units for new time scale, one of TIME_UNITS

Returns

MultiF0Data – the resampled annotation

to_matrix(*time_scale*, *time_scale_unit*, *frequency_scale*, *frequency_scale_unit*, *amplitude_unit='binary'*)

Convert f0 data to a matrix (piano roll) defined by a time and frequency scale

Parameters

- **time_scale** (*np.array*) – times in units time_unit
- **time_scale_unit** (*str*) – time scale units, one of TIME_UNITS
- **frequency_scale** (*np.array*) – frequencies in frequency_unit
- **frequency_scale_unit** (*str*) – frequency scale units, one of PITCH_UNITS
- **amplitude_unit** (*str*) – amplitude units, one of AMPLITUDE_UNITS Defaults to "binary".

Returns

np.ndarray – 2D matrix of shape len(time_scale) x len(frequency_scale)

to_mir_eval()

Convert annotation into the format expected by mir_eval.multipitch.evaluate

Returns

** times (np.ndarray)* – array of uniformly spaced time stamps in seconds * frequency_list (list): list of np.array of frequency values in Hz

to_sparse_index(*time_scale*, *time_scale_unit*, *frequency_scale*, *frequency_scale_unit*, *amplitude_unit='binary'*)

Convert MultiF0 annotation to sparse matrix indices for a time-frequency matrix.

Parameters

- **time_scale** (*np.array*) – times in units time_unit
- **time_scale_unit** (*str*) – time scale units, one of TIME_UNITS
- **frequency_scale** (*np.array*) – frequencies in frequency_unit
- **frequency_scale_unit** (*str*) – frequency scale units, one of PITCH_UNITS
- **amplitude_unit** (*str*) – amplitude units, one of AMPLITUDE_UNITS Defaults to "binary".

Returns

** sparse_index (np.ndarray)* – Array of sparce indices [(time_index, frequency_index)] * amplitude (np.ndarray): Array of amplitude values for each index

class mirdata.annotations.**NoteData**(*intervals: numpy.ndarray*, *interval_unit: str*, *pitches: numpy.ndarray*, *pitch_unit: str*, *confidence: numpy.ndarray | None = None*, *confidence_unit: str | None = None*)

NoteData class

Variables

- **intervals** (*np.ndarray*) – (n x 2) array of intervals in the form [start_time, end_time]. Times should be positive and intervals should have non-negative duration
- **interval_unit** (*str*) – unit of the time values in intervals. One of TIME_UNITS.
- **pitches** (*np.ndarray*) – array of pitches
- **pitch_unit** (*str*) – note unit, one of PITCH_UNITS
- **confidence** (*np.ndarray or None*) – array of confidence values
- **confidence_unit** (*str or None*) – confidence unit, one of AMPLITUDE_UNITS

to_matrix(*time_scale: numpy.ndarray*, *time_scale_unit: str*, *frequency_scale: numpy.ndarray*, *frequency_scale_unit: str*, *amplitude_unit: str = 'binary'*, *onsets_only: bool = False*) → numpy.ndarray

Convert f0 data to a matrix (piano roll) defined by a time and frequency scale

Parameters

- **time_scale** (*np.ndarray*) – array of matrix time stamps in seconds
- **time_scale_unit** (*str*) – units for time scale values, one of TIME_UNITS
- **frequency_scale** (*np.ndarray*) – array of matrix frequency values in seconds
- **frequency_scale_unit** (*str*) – units for frequency scale values, one of PITCH_UNITS
- **onsets_only** (*bool, optional*) – If True, returns an onset piano roll. Defaults to False.

Returns

np.ndarray – 2D matrix of shape len(time_scale) x len(frequency_scale)

to_mir_eval()

Convert data to the format expected by mir_eval.transcription.evaluate and mir_eval.transcription_velocity.evaluate

Returns

- intervals (np.ndarray) - (n x 2) array of intervals of start time, end time in seconds
- pitches (np.ndarray) - array of pitch values in hz
- velocity (optional, np.ndarray) - array of velocity values between 0 and 127

to_multif0(*time_hop: float*, *time_hop_unit: str*, *max_time: float | None = None*) → *MultiF0Data*

Convert note annotation to multiple f0 format.

Parameters

- **time_hop** (*float*) – time between time stamps in multif0 annotation
- **time_hop_unit** (*str*) – unit for time_hop, and resulting multif0 data. One of TIME_UNITS
- **max_time** (*float, optional*) – Maximum time stamp in time_hop units. Defaults to None, in which case the maximum note interval time is used.

Returns

MultiF0Data – multif0 annotation

to_sparse_index(*time_scale: numpy.ndarray*, *time_scale_unit: str*, *frequency_scale: numpy.ndarray*, *frequency_scale_unit: str*, *amplitude_unit: str = 'binary'*, *onsets_only: bool = False*) → Tuple[numpy.ndarray, numpy.ndarray]

Convert note annotations to indexes of a sparse matrix (piano roll)

Parameters

- **time_scale** (*np.array*) – array of matrix time stamps in seconds
- **time_scale_unit** (*str*) – units for time scale values, one of TIME_UNITS
- **frequency_scale** (*np.array*) – array of matrix frequency values in seconds
- **frequency_scale_unit** (*str*) – units for frequency scale values, one of PITCH_UNITS
- **amplitude_unit** (*str*) – units for amplitude values, one of AMPLITUDE_UNITS. Defaults to "binary".
- **onsets_only** (*bool, optional*) – If True, returns an onset piano roll. Defaults to False.

Returns

** sparse_index (np.ndarray)* – Array of sparce indices [(time_index, frequency_index)] * amplitude (np.ndarray): Array of amplitude values for each index

mirdata.annotations.**PITCH_UNITS = {'hz': 'hertz', 'midi': 'MIDI note number', 'note_name': 'pc with octave, e.g. Ab4', 'pc': 'pitch class, e.g. G#'}**

Pitch units

mirdata.annotations.**SECTION_UNITS = {'open': 'no scrict schema or units'}**

Section units

class mirdata.annotations.**SectionData**(*intervals*, *interval_unit*, *labels=None*, *label_unit=None*)

SectionData class

Variables

- **intervals** (*np.ndarray*) – (n x 2) array of intervals in the form [start_time, end_time]. Times should be positive and intervals should have non-negative duration
- **interval_unit** (*str*) – unit of the time values in intervals. One of TIME_UNITS.
- **labels** (*list or None*) – list of section labels
- **label_unit** (*str or None*) – label unit, one of SECTION_UNITS

mirdata.annotations.**TEMPO_UNITS = {'bpm': 'beats per minute'}**

Tempo units

mirdata.annotations.**TIME_UNITS = {'ms': 'miliseconds', 's': 'seconds', 'ticks': 'MIDI ticks'}**

Time units

class mirdata.annotations.**TempoData**(*intervals*, *interval_unit*, *tempos*, *tempo_unit*, *confidence=None*, *confidence_unit=None*)

TempoData class

Variables

- **intervals** (*np.ndarray*) – (n x 2) array of intervals in the form [start_time, end_time]. Times should be positive and intervals should have non-negative duration
- **interval_unit** (*str*) – unit of the time values in intervals. One of TIME_UNITS.
- **tempos** (*list*) – array of tempo values (as floats)
- **tempo_unit** (*str*) – tempo unit, one of TEMPO_UNITS
- **confidence** (*np.ndarray or None*) – array of confidence values
- **confidence_unit** (*str or None*) – confidence unit, one of AMPLITUDE_UNITS

mirdata.annotations.**VOICING_UNITS = {'binary': '0 or 1', 'likelihood': 'score between 0 and 1'}**

Voicing units

mirdata.annotations.**closest_index**(*input_array*, *target_array*)

Get array of indices of target_array that are closest to the input_array

Parameters

- **input_array** (*np.ndarray*) – (n x 2) array of input values
- **target_array** (*np.ndarray*) – (m x 2) array of target values)

Returns

np.ndarray – array of shape (n x 1) of indexes into target_array

mirdata.annotations.**convert_amplitude_units**(*amplitude*, *amplitude_unit*, *target_amplitude_unit*)

Convert amplitude values to likelihoods

Parameters

- **amplitude** (*np.array*) – array of amplitude values
- **amplitude_unit** (*str*) – unit of amplitude, one of AMPLITUDE_UNITS
- **target_amplitude_unit** (*str*) – target unit of amplitude, one of AMPLITUDE_UNITS

Raises

NotImplementedError – If conversion is not supported

Returns

np.array – array of amplitude values as in target amplitude unit

mirdata.annotations.**convert_pitch_units**(*pitches*, *pitch_unit*, *target_pitch_unit*)

Convert pitch values from pitch_unit to target_pitch_unit

Parameters

- **pitches** (*np.array*) – array of pitch values
- **pitch_unit** (*str*) – unit of pitch, one of PITCH_UNITS

- **target_pitch_unit** (*str*) – target unit of pitch, one of PITCH_UNITS

Raises

NotImplementedError – If conversion between given units is not supported

Returns

np.array – array of pitch values in target_pitch_unit

mirdata.annotations.**convert_time_units**(*times*, *time_unit*, *target_time_unit*)

Convert a time array from time_unit to target_time_unit

Parameters

- **times** (*np.ndarray*) – array of time values in units time_unit
- **time_unit** (*str*) – time unit, one of TIME_UNITS
- **target_time_unit** (*str*) – new time unit, one of TIME_UNITS

Raises

ValueError – If time units are not convertable

Returns

np.ndarray – times in units target_time_unit

mirdata.annotations.**validate_array_like**(*array_like*, *expected_type*, *expected_dtype*, *none_allowed=False*)

Validate that array-like object is well formed

If array_like is None, validation passes automatically.

Parameters

- **array_like** (*array-like*) – object to validate
- **expected_type** (*type*) – expected type, either list or np.ndarray
- **expected_dtype** (*type*) – expected dtype
- **none_allowed** (*bool*) – if True, allows array to be None

Raises

- **TypeError** – if type/dtype does not match expected_type/expected_dtype
- **ValueError** – if array

mirdata.annotations.**validate_beat_positions**(*positions*, *position_unit*)

Validate if positions is well-formed.

Parameters

- **positions** (*np.ndarray*) – an array of positions values
- **positions_unit** (*str*) – one of BEAT_POSITION_UNITS

Raises

ValueError – if positions values are incompatible with the unit

mirdata.annotations.**validate_chord_labels**(*chords*, *chord_unit*)

Validate that chord labels conform to chord_unit namespace

Parameters

- **chords** (*list*) – list of chord labels as strings
- **chord_unit** (*str*) – chord namespace, e.g. “harte”

Raises
ValueError – If chords don't conform to namespace

mirdata.annotations.**validate_confidence**(*confidence*, *confidence_unit*)

Validate if confidence is well-formed.

If confidence is None, validation passes automatically

Parameters

- **confidence** (*np.ndarray*) – an array of confidence values
- **confidence_unit** (*str*) – one of AMPLITUDE_UNITS

Raises
ValueError – if confidence values are incompatible with the unit

mirdata.annotations.**validate_intervals**(*intervals*, *interval_unit*)

Validate if intervals are well-formed.

If intervals is None, validation passes automatically

Parameters

- **intervals** (*np.ndarray*) – (n x 2) array
- **interval_unit** (*str*) – interval unit, one of TIME_UNITS

Raises

- **ValueError** – if intervals have an invalid shape, have negative values
- **or if end times are smaller than start times.** –

mirdata.annotations.**validate_key_labels**(*keys*, *key_unit*)

Validate that key labels conform to key_unit namespace

Parameters

- **keys** (*list*) – list of key labels as strings
- **key_unit** (*str*) – key namespace, e.g. "harte"

Raises
ValueError – If keys don't conform to namespace

mirdata.annotations.**validate_lengths_equal**(*array_list*)

Validate that arrays in list are equal in length

Some arrays may be None, and the validation for these are skipped.

Parameters
array_list (*list*) – list of array-like objects

Raises
ValueError – if arrays are not equal in length

mirdata.annotations.**validate_pitches**(*pitches*, *pitch_unit*)

Validate if pitches are well-formed.

Parameters

- **pitches** (*np.ndarray*) – an array of pitch values
- **pitch_unit** (*str*) – pitch unit, one of PITCH_UNITS

Raises
ValueError – if pitches do not correspond to the unit

mirdata.annotations.**validate_tempos**(*tempo*, *tempo_unit*)

Validate if tempos are well-formed

Parameters

- **tempo** (*list*) – list of tempo values
- **tempo_unit** (*str*) – tempo unit, one of TEMPO_UNITS

Raises
ValueError – if tempos are not well-formed

mirdata.annotations.**validate_times**(*times*, *time_unit*)

Validate if times are well-formed.

If times is None, validation passes automatically

Parameters

- **times** (*np.ndarray*) – an array of time stamps
- **time_unit** (*str*) – one of TIME_UNITS

Raises
ValueError – if times have negative values or are non-increasing

mirdata.annotations.**validate_unit**(*unit*, *unit_values*, *allow_none=False*)

Validate that the given unit is one of the allowed unit values.

Parameters

- **unit** (*str*) – the unit name
- **unit_values** (*dict*) – dictionary of possible unit values
- **allow_none** (*bool*) – if true, allows unit=None to pass validation

Raises
ValueError – If the given unit is not one of the allowed unit valuess

mirdata.annotations.**validate_voicing**(*voicing*, *voicing_unit*)

Validate if voicing is well-formed.

Parameters

- **voicing** (*np.ndarray*) – an array of voicing values
- **voicing_unit** (*str*) – one of VOICING_UNITS

Raises
ValueError – if voicing values are incompatible with the unit

2.8 Advanced

2.8.1 mirdata.validate

Utility functions for mirdata

mirdata.validate.**log_message**(*message*, *verbose=True*)

Helper function to log message

Parameters

- **message** (*str*) – message to log
- **verbose** (*bool*) – if false, the message is not logged

mirdata.validate.**md5**(*file_path*)

Get md5 hash of a file.

Parameters

file_path (*str*) – File path

Returns

str – md5 hash of data in file_path

mirdata.validate.**validate**(*local_path*, *checksum*)

Validate that a file exists and has the correct checksum

Parameters

- **local_path** (*str*) – file path
- **checksum** (*str*) – md5 checksum

Returns

- bool - True if file exists
- bool - True if checksum matches

mirdata.validate.**validate_files**(*file_dict*, *data_home*, *verbose*)

Validate files

Parameters

- **file_dict** (*dict*) – dictionary of file information
- **data_home** (*str*) – path where the data lives
- **verbose** (*bool*) – if True, show progress

Returns

- dict - missing files
- dict - files with invalid checksums

mirdata.validate.**validate_index**(*dataset_index*, *data_home*, *verbose=True*)

Validate files in a dataset's index

Parameters

- **dataset_index** (*list*) – dataset indices
- **data_home** (*str*) – Local home path that the dataset is being stored

- **verbose** (*bool*) – if true, prints validation status while running

Returns

- dict - file paths that are in the index but missing locally
- dict - file paths with differing checksums

mirdata.validate.**validate_metadata**(*file_dict*, *data_home*, *verbose*)

Validate files

Parameters

- **file_dict** (*dict*) – dictionary of file information
- **data_home** (*str*) – path where the data lives
- **verbose** (*bool*) – if True, show progress

Returns

- dict - missing files
- dict - files with invalid checksums

mirdata.validate.**validator**(*dataset_index*, *data_home*, *verbose=True*)

Checks the existence and validity of files stored locally with respect to the paths and file checksums stored in the reference index. Logs invalid checksums and missing files.

Parameters

- **dataset_index** (*list*) – dataset indices
- **data_home** (*str*) – Local home path that the dataset is being stored
- **verbose** (*bool*) – if True (default), prints missing and invalid files to stdout. Otherwise, this function is equivalent to validate_index.

Returns

missing_files (list) –

List of file paths that are in the dataset index
but missing locally.

invalid_checksums (list): List of file paths that file exists in the
dataset index but has a different checksum compare to the reference checksum.

2.8.2 mirdata.download_utils

Utilities for downloading from the web.

class mirdata.download_utils.**DownloadProgressBar**(*_, **__)

Wrap *tqdm* to show download progress

class mirdata.download_utils.**RemoteFileMetadata**(*filename*, *url*, *checksum*, *destination_dir=None*, *unpack_directories=None*)

The metadata for a remote file

Variables

- **filename** (*str*) – the remote file's basename
- **url** (*str*) – the remote file's url

- **checksum** (*str*) – the remote file's md5 checksum
- **destination_dir** (*str or None*) – the relative path for where to save the file
- **unpack_directories** (*list or None*) – list of relative directories. For each directory the contents will be moved to destination_dir (or data_home if not provieds)

mirdata.download_utils.**download_from_remote**(*remote*, *save_dir*, *force_overwrite*, *allow_invalid_checksum*)

Download a remote dataset into path Fetch a dataset pointed by remote's url, save into path using remote's filename and ensure its integrity based on the MD5 Checksum of the downloaded file.

Adapted from scikit-learn's sklearn.datasets.base._fetch_remote.

Parameters

- **remote** (*RemoteFileMetadata*) – Named tuple containing remote dataset meta information: url, filename and checksum
- **save_dir** (*str*) – Directory to save the file to. Usually *data_home*
- **force_overwrite** (*bool*) – If True, overwrite existing file with the downloaded file. If False, does not overwrite, but checks that checksum is consistent.

Returns

str – Full path of the created file.

mirdata.download_utils.**download_tar_file**(*tar_remote*, *save_dir*, *force_overwrite*, *cleanup*, *allow_invalid_checksum*)

Download and untar a tar file.

Parameters

- **tar_remote** (*RemoteFileMetadata*) – Object containing download information
- **save_dir** (*str*) – Path to save downloaded file
- **force_overwrite** (*bool*) – If True, overwrites existing files
- **cleanup** (*bool*) – If True, remove tarfile after untarring

mirdata.download_utils.**download_zip_file**(*zip_remote*, *save_dir*, *force_overwrite*, *cleanup*, *allow_invalid_checksum*)

Download and unzip a zip file.

Parameters

- **zip_remote** (*RemoteFileMetadata*) – Object containing download information
- **save_dir** (*str*) – Path to save downloaded file
- **force_overwrite** (*bool*) – If True, overwrites existing files
- **cleanup** (*bool*) – If True, remove zipfile after unziping

mirdata.download_utils.**downloader**(*save_dir*, *remotes=None*, *index=None*, *partial_download=None*, *info_message=None*, *force_overwrite=False*, *cleanup=False*, *allow_invalid_checksum=False*)

Download data to *save_dir* and optionally log a message.

Parameters

- **save_dir** (*str*) – The directory to download the data

- **remotes** (*dict or None*) – A dictionary of RemoteFileMetadata tuples of data in zip format. If None, there is no data to download
- **index** (*core.Index*) – A mirdata Index class, which may contain a remote index to be downloaded or a subset of remotes to download by default.
- **partial_download** (*list or None*) – A list of keys to partially download the remote objects of the download dict. If None, all data specified by the index is downloaded
- **info_message** (*str or None*) – A string of info to log when this function is called. If None, no string is logged.
- **force_overwrite** (*bool*) – If True, existing files are overwritten by the downloaded files.
- **cleanup** (*bool*) – Whether to delete the zip/tar file after extracting.
- **allow_invalid_checksum** (*bool*) – Allow having an invalid checksum, and whenever this happens prompt a warning instead of deleting the files.

mirdata.download_utils.**extractall_unicode**(*zfile*, *out_dir*)

Extract all files inside a zip archive to a output directory.

In comparison to the zipfile, it checks for correct file name encoding

Parameters

- **zfile** (*obj*) – Zip file object created with zipfile.ZipFile
- **out_dir** (*str*) – Output folder

mirdata.download_utils.**move_directory_contents**(*source_dir*, *target_dir*)

Move the contents of source_dir into target_dir, and delete source_dir

Parameters

- **source_dir** (*str*) – path to source directory
- **target_dir** (*str*) – path to target directory

mirdata.download_utils.**untar**(*tar_path*, *cleanup*)

Untar a tar file inside it's current directory.

Parameters

- **tar_path** (*str*) – Path to tar file
- **cleanup** (*bool*) – If True, remove tarfile after untarring

mirdata.download_utils.**unzip**(*zip_path*, *cleanup*)

Unzip a zip file inside it's current directory.

Parameters

- **zip_path** (*str*) – Path to zip file
- **cleanup** (*bool*) – If True, remove zipfile after unzipping

2.8.3 mirdata.jams_utils

Utilities for converting mirdata Annotation classes to jams format.

mirdata.jams_utils.**beats_to_jams**(*beat_data*, *description=None*)

Convert beat annotations into jams format.

Parameters

- **beat_data** (*annotations.BeatData*) – beat data object
- **description** (*str*) – annotation description

Returns

jams.Annotation – jams annotation object.

mirdata.jams_utils.**chords_to_jams**(*chord_data*, *description=None*)

Convert chord annotations into jams format.

Parameters

- **chord_data** (*annotations.ChordData*) – chord data object
- **description** (*str*) – annotation description

Returns

jams.Annotation – jams annotation object.

mirdata.jams_utils.**events_to_jams**(*event_data*, *description=None*)

Convert events annotations into jams format.

Parameters

- **event_data** (*annotations.EventData*) – event data object
- **description** (*str*) – annotation description

Returns

jams.Annotation – jams annotation object.

mirdata.jams_utils.**f0s_to_jams**(*f0_data*, *description=None*)

Convert f0 annotations into jams format.

Parameters

- **f0_data** (*annotations.F0Data*) – f0 annotation object
- **description** (*str*) – annotation descriptoin

Returns

jams.Annotation – jams annotation object.

mirdata.jams_utils.**jams_converter**(*audio_path=None*, *spectrogram_path=None*, *beat_data=None*, *chord_data=None*, *note_data=None*, *f0_data=None*, *section_data=None*, *multi_section_data=None*, *tempo_data=None*, *event_data=None*, *key_data=None*, *lyrics_data=None*, *tags_gtzan_data=None*, *tags_open_data=None*, *metadata=None*)

Convert annotations from a track to JAMS format.

Parameters

- **audio_path** (*str or None*) – A path to the corresponding audio file, or None. If provided, the audio file will be read to compute the duration. If None, 'duration' must be a field in the metadata dictionary, or the resulting jam object will not validate.

- **spectrogram_path** (*str or None*) – A path to the corresponding spectrum file, or None.
- **beat_data** (*list or None*) – A list of tuples of (annotations.BeatData, str), where str describes the annotation (e.g. 'beats_1').
- **chord_data** (*list or None*) – A list of tuples of (annotations.ChordData, str), where str describes the annotation.
- **note_data** (*list or None*) – A list of tuples of (annotations.NoteData, str), where str describes the annotation.
- **f0_data** (*list or None*) – A list of tuples of (annotations.F0Data, str), where str describes the annotation.
- **section_data** (*list or None*) – A list of tuples of (annotations.SectionData, str), where str describes the annotation.
- **multi_section_data** (*list or None*) – A list of tuples. Tuples in multi_section_data should contain another list of tuples, indicating annotations in the different levels e.g. ([(segments0, level0), '(segments1, level1)], annotator) and a str indicating the annotator
- **tempo_data** (*list or None*) – A list of tuples of (float, str), where float gives the tempo in bpm and str describes the annotation.
- **event_data** (*list or None*) – A list of tuples of (annotations.EventData, str), where str describes the annotation.
- **key_data** (*list or None*) – A list of tuples of (annotations.KeyData, str), where str describes the annotation.
- **lyrics_data** (*list or None*) – A list of tuples of (annotations.LyricData, str), where str describes the annotation.
- **tags_gtzan_data** (*list or None*) – A list of tuples of (str, str), where the first srt is the tag and the second is a descriptor of the annotation.
- **tags_open_data** (*list or None*) – A list of tuples of (str, str), where the first srt is the tag and the second is a descriptor of the annotation.
- **metadata** (*dict or None*) – A dictionary containing the track metadata.

Returns

jams.JAMS – A JAMS object containing the annotations.

mirdata.jams_utils.**keys_to_jams**(*key_data*, *description*)

Convert key annotations into jams format.

Parameters

- **key_data** (*annotations.KeyData*) – key data object
- **description** (*str*) – annotation description

Returns

jams.Annotation – jams annotation object.

mirdata.jams_utils.**lyrics_to_jams**(*lyric_data*, *description=None*)

Convert lyric annotations into jams format.

Parameters

- **lyric_data** (*annotations.LyricData*) – lyric annotation object
- **description** (*str*) – annotation descriptoin

Returns
jams.Annotation – jams annotation object.

mirdata.jams_utils.**multi_sections_to_jams**(*multisection_data*, *description*)

Convert multi-section annotations into jams format.

Parameters

- **multisection_data** (*list*) – list of tuples of the form [(SectionData, int)]
- **description** (*str*) – annotation description

Returns
jams.Annotation – jams annotation object.

mirdata.jams_utils.**notes_to_jams**(*note_data*, *description*)

Convert note annotations into jams format.

Parameters

- **note_data** (*annotations.NoteData*) – note data object
- **description** (*str*) – annotation description

Returns
jams.Annotation – jams annotation object.

mirdata.jams_utils.**sections_to_jams**(*section_data*, *description=None*)

Convert section annotations into jams format.

Parameters

- **section_data** (*annotations.SectionData*) – section data object
- **description** (*str*) – annotation description

Returns
jams.Annotation – jams annotation object.

mirdata.jams_utils.**tag_to_jams**(*tag_data*, *namespace='tag_open'*, *description=None*)

Convert lyric annotations into jams format.

Parameters

- **lyric_data** (*annotations.LyricData*) – lyric annotation object
- **namespace** (*str*) – the jams-compatible tag namespace
- **description** (*str*) – annotation descriptoin

Returns
jams.Annotation – jams annotation object.

mirdata.jams_utils.**tempos_to_jams**(*tempo_data*, *description=None*)

Convert tempo annotations into jams format.

Parameters

- **tempo_data** (*annotations.TempoData*) – tempo data object
- **description** (*str*) – annotation description

Returns
jams.Annotation – jams annotation object.

2.9 Contributing

We encourage contributions to mirdata, especially new dataset loaders. To contribute a new loader, follow the steps indicated below and create a Pull Request (PR) to the github repository. For any doubt or comment about your contribution, you can always submit an issue or open a discussion in the repository.

- Issue Tracker
- Source Code

To reduce friction, we may make commits on top of contributor's PRs. If you do not want us to, please tag your PR with `please-do-not-edit`.

2.9.1 Installing mirdata for development purposes

To install `mirdata` for development purposes:

- First run:

```
git clone https://github.com/mir-dataset-loaders/mirdata.git
```

- Then, after opening source data library you have to install the dependencies for updating the documentation and running tests:

```
pip install .
pip install ."[tests]"
pip install ."[docs]"
pip install ."[dali]"
pip install ."[haydn_op20]"
```

We recommend to install pyenv to manage your Python versions and install all `mirdata` requirements. You will want to install the latest supported Python versions (see README.md). Once `pyenv` and the Python versions are configured, install `pytest`. Make sure you installed all the necessary pytest plugins to automatically test your code successfully (e.g. *pytest-cov*). Finally, run:

Before running the tests, make sure to have formatted `mirdata/` and `tests/` with `black`.

```
black mirdata/ tests/
```

Also, make sure that they pass flake8 and mypy tests specified in lint-python.yml github action workflow.

```
flake8 mirdata --count --select=E9,F63,F7,F82 --show-source --statistics
python -m mypy mirdata --ignore-missing-imports --allow-subclassing-any
```

Finally, run:

```
pytest -vv --cov-report term-missing --cov-report=xml --cov=mirdata tests/ --local
```

All tests should pass!

2.9.2 Writing a new dataset loader

The steps to add a new dataset loader to `mirdata` are:

1. *Create an index*
2. *Create a module*
3. *Add tests*
4. *Submit your loader*

Before starting, check if your dataset falls into one of these non-standard cases:

- Is the dataset not freely downloadable? If so, see *this section*
- Does the dataset require dependencies not currently in mirdata? If so, see *this section*
- Does the dataset have multiple versions? If so, see *this section*
- Is the index large (e.g. > 5 MB)? If so, see *this section*

1. Create an index

`mirdata`'s structure relies on *indexes*. Indexes are dictionaries contain information about the structure of the dataset which is necessary for the loading and validating functionalities of `mirdata`. In particular, indexes contain information about the files included in the dataset, their location and checksums. The necessary steps are:

1. To create an index, first create a script in `scripts/`, as `make_dataset_index.py`, which generates an index file.
2. Then run the script on the dataset and save the index in `mirdata/datasets/indexes/` as `dataset_index_<version>.json`. where <version> indicates which version of the dataset was used (e.g. 1.0).

Here there is an example of an index to use as guideline:

Example Make Index Script

```
import argparse
import glob
import json
import os
from mirdata.validate import md5

DATASET_INDEX_PATH = "../mirdata/datasets/indexes/dataset_index.json"


def make_dataset_index(dataset_data_path):
    annotation_dir = os.path.join(dataset_data_path, "annotation")
    annotation_files = glob.glob(os.path.join(annotation_dir, "*.lab"))
    track_ids = sorted([os.path.basename(f).split(".")[0] for f in annotation_files])

    # top-key level metadata
    metadata_checksum = md5(os.path.join(dataset_data_path, "id_mapping.txt"))
    index_metadata = {"metadata": {"id_mapping": ("id_mapping.txt", metadata_checksum)}}

    # top-key level tracks
```

(continues on next page)

(continued from previous page)

```
    index_tracks = {}
    for track_id in track_ids:
        audio_checksum = md5(
            os.path.join(dataset_data_path, "Wavfile/{}.wav".format(track_id))
        )
        annotation_checksum = md5(
            os.path.join(dataset_data_path, "annotation/{}.lab".format(track_id))
        )

        index_tracks[track_id] = {
            "audio": ("Wavfile/{}.wav".format(track_id), audio_checksum),
            "annotation": ("annotation/{}.lab".format(track_id), annotation_checksum),
        }

    # top-key level version
    dataset_index = {"version": None}

    # combine all in dataset index
    dataset_index.update(index_metadata)
    dataset_index.update({"tracks": index_tracks})

    with open(DATASET_INDEX_PATH, "w") as fhandle:
        json.dump(dataset_index, fhandle, indent=2)


def main(args):
    make_dataset_index(args.dataset_data_path)


if __name__ == "__main__":
    PARSER = argparse.ArgumentParser(description="Make dataset index file.")
    PARSER.add_argument(
        "dataset_data_path", type=str, help="Path to dataset data folder."
    )

    main(PARSER.parse_args())
```

More examples of scripts used to create dataset indexes can be found in the scripts folder.

tracks

Most MIR datasets are organized as a collection of tracks and annotations. In such case, the index should make use of the `tracks` top-level key. A dictionary should be stored under the `tracks` top-level key where the keys are the unique track ids of the dataset. The values are a dictionary of files associated with a track id, along with their checksums. These files can be for instance audio files or annotations related to the track id. File paths are relative to the top level directory of a dataset.

Index Examples - Tracks

If the version *1.0* of a given dataset has the structure:

```
> Example_Dataset/
    > audio/
        track1.wav
        track2.wav
        track3.wav
    > annotations/
        track1.csv
        Track2.csv
        track3.csv
    > metadata/
        metadata_file.csv
```

The top level directory is `Example_Dataset` and the relative path for `track1.wav` would be `audio/track1.wav`. Any unavailable fields are indicated with *null*. A possible index file for this example would be:

```
{   "version": "1.0",
    "tracks":
        "track1": {
            "audio": [
                "audio/track1.wav",  // the relative path for track1's audio file
                "912ec803b2ce49e4a541068d495ab570"  // track1.wav's md5 checksum
            ],
            "annotation": [
                "annotations/track1.csv",  // the relative path for track1's annotation
                "2cf33591c3b28b382668952e236cccd5"  // track1.csv's md5 checksum
            ]
        },
        "track2": {
            "audio": [
                "audio/track2.wav",
                "65d671ec9787b32cfb7e33188be32ff7"
            ],
            "annotation": [
                "annotations/Track2.csv",
                "e1964798cfe86e914af895f8d0291812"
            ]
        },
        "track3": {
            "audio": [
                "audio/track3.wav",
                "60edeb51dc4041c47c031c4bfb456b76"
            ],
            "annotation": [
                "annotations/track3.csv",
                "06cb006cc7b61de6be6361ff904654b3"
            ]
        },
    }
"metadata": {
        "metadata_file": [
            "metadata/metadata_file.csv",
            "7a41b280c7b74e2ddac5184708f9525b"
        ]
```

(continues on next page)

(continued from previous page)

```
}
}
```

Note: In this example there is a (purposeful) mismatch between the name of the audio file `track2.wav` and its corresponding annotation file, `Track2.csv`, compared with the other pairs. This mismatch should be included in the index. This type of slight difference in filenames happens often in publicly available datasets, making pairing audio and annotation files more difficult. We use a fixed, version-controlled index to account for this kind of mismatch, rather than relying on string parsing on load.

multitracks

Index Examples - Multitracks

If the version *1.0* of a given multitrack dataset has the structure:

```
> Example_Dataset/
    > audio/
        multitrack1-voice1.wav
        multitrack1-voice2.wav
        multitrack1-accompaniment.wav
        multitrack1-mix.wav
        multitrack2-voice1.wav
        multitrack2-voice2.wav
        multitrack2-accompaniment.wav
        multitrack2-mix.wav
    > annotations/
        multitrack1-voice-f0.csv
        multitrack2-voice-f0.csv
        multitrack1-f0.csv
        multitrack2-f0.csv
    > metadata/
        metadata_file.csv
```

The top level directory is `Example_Dataset` and the relative path for `multitrack1-voice1` would be `audio/multitrack1-voice1.wav`. Any unavailable fields are indicated with *null*. A possible index file for this example would be:

```
{
    "version": 1,
    "tracks": {
        "multitrack1-voice": {
            "audio_voice1": ('audio/multitrack1-voice1.wav', checksum),
            "audio_voice2": ('audio/multitrack1-voice1.wav', checksum),
            "voice-f0": ('annotations/multitrack1-voice-f0.csv', checksum)
        }
        "multitrack1-accompaniment": {
            "audio_accompaniment": ('audio/multitrack1-accompaniment.wav', checksum)
        }
```

(continues on next page)

(continued from previous page)

```
        "multitrack2-voice" : {...}
        ...
    },
    "multitracks": {
        "multitrack1": {
            "tracks": ['multitrack1-voice', 'multitrack1-accompaniment'],
            "audio": ('audio/multitrack1-mix.wav', checksum)
            "f0": ('annotations/multitrack1-f0.csv', checksum)
        }
        "multitrack2": ...
    },
    "metadata": {
        "metadata_file": [
            "metadata/metadata_file.csv",
            "7a41b280c7b74e2ddac5184708f9525b"
            ]
    }
}
```

Note that in this examples we group `audio_voice1` and `audio_voice2` in a single Track because the annotation `voice-f0` annotation corresponds to their mixture. In contrast, the annotation `voice-f0` is extracted from the multitrack mix and it is stored in the `multitracks` group. The multitrack `multitrack1` has an additional track `multitrack1-mix.wav` which may be the master track, the final mix, the recording of `multitrack1` with another microphone.

records

Index Examples - Records

Coming soon

2. Create a module

Once the index is created you can create the loader. For that, we suggest you use the following template and adjust it for your dataset. To quickstart a new module:

1. Copy the example below and save it to `mirdata/datasets/<your_dataset_name>.py`
2. Find & Replace `Example` with the <your_dataset_name>.
3. Remove any lines beginning with # – which are there as guidelines.

Example Module

```
"""Example Dataset Loader

.. admonition:: Dataset Info
    :class: dropdown
```

(continues on next page)

(continued from previous page)

```
    Please include the following information at the top level docstring for the dataset's␣
↪module `dataset.py`:

    1. Describe annotations included in the dataset
    2. Indicate the size of the datasets (e.g. number files and duration, hours)
    3. Mention the origin of the dataset (e.g. creator, institution)
    4. Describe the type of music included in the dataset
    5. Indicate any relevant papers related to the dataset
    6. Include a description about how the data can be accessed and the license it uses␣
↪(if applicable)

"""
import csv
import json
import os
from typing import BinaryIO, Optional, TextIO, Tuple

# -- import whatever you need here and remove
# -- example imports you won't use
import librosa
import numpy as np
from smart_open import open  # if you use the open function, make sure you include this␣
↪line!

from mirdata import download_utils, jams_utils, core, annotations

# -- Add any relevant citations here
BIBTEX = """
@article{article-minimal,
  author = "L[eslie] B. Lamport",
  title = "The Gnats and Gnus Document Preparation System",
  journal = "G-Animal's Journal",
  year = "1986"
},
@article{article-minimal2,
  author = "L[eslie] B. Lamport",
  title = "The Gnats and Gnus Document Preparation System 2",
  journal = "G-Animal's Journal",
  year = "1987"
}
"""

# -- INDEXES specifies different versions of a dataset
# -- "default" and "test" specify which key should be used
# -- by default, and when running tests.
# -- Some datasets have a "sample" version, which is a mini-version
# -- that makes it easier to try out a large dataset without needing
# -- to download the whole thing.
# -- If there is no sample version, simply set "test": "1.0".
# -- If the default data is remote, there must be a local sample for tests!
INDEXES = {
    "default": "1.0",
```

(continues on next page)

(continued from previous page)

```
    "test": "sample",
    "1.0": core.Index(filename="example_index_1.0.json"),
    "sample": core.Index(filename="example_index_sample.json")
}

# -- REMOTES is a dictionary containing all files that need to be downloaded.
# -- The keys should be descriptive (e.g. 'annotations', 'audio').
# -- When having data that can be partially downloaded, remember to set up
# -- correctly destination_dir to download the files following the correct structure.
REMOTES = {
    'remote_data': download_utils.RemoteFileMetadata(
        filename='a_zip_file.zip',
        url='http://website/hosting/the/zipfile.zip',
        checksum='00000000000000000000000000000000',  # -- the md5 checksum
        destination_dir='path/to/unzip' # -- relative path for where to unzip the data,␣
↪or None
    ),
}

# -- Include any information that should be printed when downloading
# -- remove this variable if you don't need to print anything during download
DOWNLOAD_INFO = """
Include any information you want to be printed when dataset.download() is called.
These can be instructions for how to download the dataset (e.g. request access on␣
↪zenodo),
caveats about the download, etc
"""

# -- Include the dataset's license information
LICENSE_INFO = """
The dataset's license information goes here.
"""


class Track(core.Track):
    """Example track class
    # -- YOU CAN AUTOMATICALLY GENERATE THIS DOCSTRING BY CALLING THE SCRIPT:
    # -- `scripts/print_track_docstring.py my_dataset`
    # -- note that you'll first need to have a test track (see "Adding tests to your␣
↪dataset" below)

    Args:
        track_id (str): track id of the track

    Attributes:
        audio_path (str): path to audio file
        annotation_path (str): path to annotation file
        # -- Add any of the dataset specific attributes here

    Cached Properties:
        annotation (EventData): a description of this annotation
```

(continues on next page)

(continued from previous page)

```
    """
    def __init__(self, track_id, data_home, dataset_name, index, metadata):

        # -- this sets the following attributes:
        # -- * track_id
        # -- * _dataset_name
        # -- * _data_home
        # -- * _track_paths
        # -- * _track_metadata
        super().__init__(
            track_id,
            data_home,
            dataset_name=dataset_name,
            index=index,
            metadata=metadata,
        )

        # -- add any dataset specific attributes here
        self.audio_path = self.get_path("audio")
        self.annotation_path = self.get_path("annotation")

        # -- if the dataset has an *official* e.g. train/test split, use this
        # -- reserved attribute (can be a property if needed)
        self.split = ...

    # -- If the dataset has metadata that needs to be accessed by Tracks,
    # -- such as a table mapping track ids to composers for the full dataset,
    # -- add them as properties like instead of in the __init__.
    @property
    def composer(self) -> Optional[str]:
        return self._track_metadata.get("composer")

    # -- `annotation` will behave like an attribute, but it will only be loaded
    # -- and saved when someone accesses it. Useful when loading slightly
    # -- bigger files or for bigger datasets. By default, we make any time
    # -- series data loaded from a file a cached property
    @core.cached_property
    def annotation(self) -> Optional[annotations.EventData]:
        return load_annotation(self.annotation_path)

    # -- `audio` will behave like an attribute, but it will only be loaded
    # -- when someone accesses it and it won't be stored. By default, we make
    # -- any memory heavy information (like audio) properties
    @property
    def audio(self) -> Optional[Tuple[np.ndarray, float]]:
        """The track's audio

        Returns:
            * np.ndarray - audio signal
            * float - sample rate

        """
```

(continues on next page)

(continued from previous page)

```
        return load_audio(self.audio_path)

    # -- we use the to_jams function to convert all the annotations in the JAMS format.
    # -- The converter takes as input all the annotations in the proper format (e.g. beats
    # -- will be fed as beat_data=[(self.beats, None)], see jams_utils), and returns a jams
    # -- object with the annotations.
    def to_jams(self):
        """Jams: the track's data in jams format"""
        return jams_utils.jams_converter(
            audio_path=self.audio_path,
            annotation_data=[(self.annotation, None)],
            metadata=self._metadata,
        )
        # -- see the documentation for `jams_utils.jams_converter for all fields


# -- if the dataset contains multitracks, you can define a MultiTrack similar to a Track
# -- you can delete the block of code below if the dataset has no multitracks
class MultiTrack(core.MultiTrack):
    """Example multitrack class

    Args:
        mtrack_id (str): multitrack id
        data_home (str): Local path where the dataset is stored.
            If `None`, looks for the data in the default directory, `~/mir_datasets/Example`

    Attributes:
        mtrack_id (str): track id
        tracks (dict): {track_id: Track}
        track_audio_property (str): the name of the attribute of Track which
            returns the audio to be mixed
        # -- Add any of the dataset specific attributes here

    Cached Properties:
        annotation (EventData): a description of this annotation

    """
    def __init__(
        self, mtrack_id, data_home, dataset_name, index, track_class, metadata
    ):
        # -- this sets the following attributes:
        # -- * mtrack_id
        # -- * _dataset_name
        # -- * _data_home
        # -- * _multitrack_paths
        # -- * _metadata
        # -- * _track_class
        # -- * _index
        # -- * track_ids
```

(continues on next page)

(continued from previous page)

```
        super().__init__(
            mtrack_id=mtrack_id,
            data_home=data_home,
            dataset_name=dataset_name,
            index=index,
            track_class=track_class,
            metadata=metadata,
        )

        # -- optionally add any multitrack specific attributes here
        self.mix_path = ...  # this can be called whatever makes sense for the datasets
        self.annotation_path = ...

        # -- if the dataset has an *official* e.g. train/test split, use this
        # -- reserved attribute (can be a property if needed)
        self.split = ...

    # If you want to support multitrack mixing in this dataset, set this property
    @property
    def track_audio_property(self):
        return "audio"  # the attribute of Track, e.g. Track.audio, which returns the␣
↪audio to mix

    # -- multitracks can optionally have mix-level cached properties and properties
    @core.cached_property
    def annotation(self) -> Optional[annotations.EventData]:
        """output type: description of output"""
        return load_annotation(self.annotation_path)

    @property
    def audio(self) -> Optional[Tuple[np.ndarray, float]]:
        """The track's audio

        Returns:
            * np.ndarray - audio signal
            * float - sample rate

        """
        return load_audio(self.audio_path)

    # -- multitrack classes are themselves Tracks, and also need a to_jams method
    # -- for any mixture-level annotations
    def to_jams(self):
        """Jams: the track's data in jams format"""
        return jams_utils.jams_converter(
            audio_path=self.mix_path,
            annotation_data=[(self.annotation, None)],
            ...
        )
        # -- see the documentation for `jams_utils.jams_converter for all fields
```

(continues on next page)

(continued from previous page)

```
# -- this decorator allows this function to take a string or an open bytes file as input
# -- and in either case converts it to an open file handle.
# -- It also checks if the file exists
# -- and, if None is passed, None will be returned
@io.coerce_to_bytes_io
def load_audio(fhandle: BinaryIO) -> Tuple[np.ndarray, float]:
    """Load a Example audio file.

    Args:
        fhandle (str or file-like): path or file-like object pointing to an audio file

    Returns:
        * np.ndarray - the audio signal
        * float - The sample rate of the audio file

    """
    # -- for example, the code below. This should be dataset specific!
    # -- By default we load to mono
    # -- change this if it doesn't make sense for your dataset.
    return librosa.load(audio_path, sr=None, mono=True)


# -- Write any necessary loader functions for loading the dataset's data

# -- this decorator allows this function to take a string or an open file as input
# -- and in either case converts it to an open file handle.
# -- It also checks if the file exists
# -- and, if None is passed, None will be returned
@io.coerce_to_string_io
def load_annotation(fhandle: TextIO) -> Optional[annotations.EventData]:

    # -- because of the decorator, the file is already open
    reader = csv.reader(fhandle, delimiter=' ')
    intervals = []
    annotation = []
    for line in reader:
        intervals.append([float(line[0]), float(line[1])])
        annotation.append(line[2])

    # there are several annotation types in annotations.py
    # They should be initialized with data, followed by their units
    # see annotations.py for a complete list of types and units.
    annotation_data = annotations.EventData(
        np.array(intervals), "s", np.array(annotation), "open"
    )
    return annotation_data

# -- use this decorator so the docs are complete
@core.docstring_inherit(core.Dataset)
class Dataset(core.Dataset):
    """The Example dataset
    """
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, data_home=None, version="default"):
        super().__init__(
            data_home,
            version,
            name=NAME,
            track_class=Track,
            bibtex=BIBTEX,
            indexes=INDEXES,
            remotes=REMOTES,
            download_info=DOWNLOAD_INFO,
            license_info=LICENSE_INFO,
        )

    # -- if your dataset has a top-level metadata file, write a loader for it here
    # -- you do not have to include this function if there is no metadata
    @core.cached_property
    def _metadata(self):
        metadata_path = os.path.join(self.data_home, 'example_metadata.csv')

        # load metadata however makes sense for your dataset
        metadata_path = os.path.join(data_home, 'example_metadata.json')
        with open(metadata_path, 'r') as fhandle:
            metadata = json.load(fhandle)

        return metadata

    # -- if your dataset needs to overwrite the default download logic, do it here.
    # -- this function is usually not necessary unless you need very custom download
↪logic
    def download(
        self, partial_download=None, force_overwrite=False, cleanup=False
    ):
        """Download the dataset

        Args:
            partial_download (list or None):
                A list of keys of remotes to partially download.
                If None, all data is downloaded
            force_overwrite (bool):
                If True, existing files are overwritten by the downloaded files.
            cleanup (bool):
                Whether to delete any zip/tar files after extracting.

        Raises:
            ValueError: if invalid keys are passed to partial_download
            IOError: if a downloaded file's checksum is different from expected

        """
        # see download_utils.downloader for basic usage - if you only need to call
↪downloader
        # once, you do not need this function at all.
```

(continues on next page)

(continued from previous page)

```
        # only write a custom function if you need it!
```

You may find these examples useful as references:

- A simple, fully downloadable dataset
- A dataset which is partially downloadable
- A dataset with restricted access data
- A dataset which uses dataset-level metadata
- A dataset which does not use dataset-level metadata
- A dataset with a custom download function
- A dataset with a remote index
- A dataset with extra dependencies
- A dataset with multitracks

For many more examples, see the datasets folder.

3. Add tests

To finish your contribution, include tests that check the integrity of your loader. For this, follow these steps:

1. Make a toy version of the dataset in the tests folder `tests/resources/mir_datasets/my_dataset/`, so you can test against little data. For example:
    - Include all audio and annotation files for one track of the dataset
    - For each audio/annotation file, reduce the audio length to 1-2 seconds and remove all but a few of the annotations.
    - If the dataset has a metadata file, reduce the length to a few lines.
2. Test all of the dataset specific code, e.g. the public attributes of the Track class, the load functions and any other custom functions you wrote. See the tests folder for reference. If your loader has a custom download function, add tests similar to this loader.
3. Locally run `pytest -s tests/test_full_dataset.py --local --dataset my_dataset` before submitting your loader to make sure everything is working. If your dataset has *multiple versions*, test each (non-default) version by running `pytest -s tests/test_full_dataset.py --local --dataset my_dataset --dataset-version my_version`.

Note: We have written automated tests for all loader's `cite`, `download`, `validate`, `load`, `track_ids` functions, as well as some basic edge cases of the `Track` class, so you don't need to write tests for these!

Example Test File

```
"""Tests for example dataset
"""
import numpy as np
```

(continues on next page)

(continued from previous page)

```
import pytest

from mirdata import annotations
from mirdata.datasets import example
from tests.test_utils import run_track_tests


def test_track():
    default_trackid = "some_id"
    data_home = "tests/resources/mir_datasets/dataset"
    dataset = example.Dataset(data_home, version="test")
    track = dataset.track(default_trackid)

    expected_attributes = {
        "track_id": "some_id",
        "audio_path": "tests/resources/mir_datasets/example/" + "Wavfile/some_id.wav",
        "song_id": "some_id",
        "annotation_path": "tests/resources/mir_datasets/example/annotation/some_id.pv",
    }

    expected_property_types = {"annotation": annotations.XData}

    assert track._track_paths == {
        "audio": ["Wavfile/some_id.wav", "278ae003cb0d323e99b9a643c0f2eeda"],
        "annotation": ["Annotation/some_id.pv", "0d93a011a9e668fd80673049089bbb14"],
    }

    run_track_tests(track, expected_attributes, expected_property_types)

    # test audio loading functions
    audio, sr = track.audio
    assert sr == 44100
    assert audio.shape == (44100 * 2,)


def test_to_jams():

    default_trackid = "some_id"
    data_home = "tests/resources/mir_datasets/dataset"
    dataset = example.Dataset(data_home, version="test")
    track = dataset.track(default_trackid)
    jam = track.to_jams()

    annotations = jam.search(namespace="annotation")[0]["data"]
    assert [annotation.time for annotation in annotations] == [0.027, 0.232]
    assert [annotation.duration for annotation in annotations] == [
        0.20500000000000002,
        0.736,
    ]
    # ... etc
```

(continues on next page)

(continued from previous page)

```
def test_load_annotation():
    # load a file which exists
    annotation_path = "tests/resources/mir_datasets/dataset/Annotation/some_id.pv"
    annotation_data = example.load_annotation(annotation_path)

    # check types
    assert type(annotation_data) == annotations.XData
    assert type(annotation_data.times) is np.ndarray
    # ... etc

    # check values
    assert np.array_equal(annotation_data.times, np.array([0.016, 0.048]))
    # ... etc


def test_metadata():
    data_home = "tests/resources/mir_datasets/dataset"
    dataset = example.Dataset(data_home, version="test")
    metadata = dataset._metadata
    assert metadata["some_id"] == "something"
```

Running your tests locally

Before creating a PR, you should run all the tests. But before that, make sure to have formatted `mirdata/` and `tests/` with `black`.

```
black mirdata/ tests/
```

Also, make sure that they pass flake8 and mypy tests specified in lint-python.yml github action workflow.

```
flake8 mirdata --count --select=E9,F63,F7,F82 --show-source --statistics
python -m mypy mirdata --ignore-missing-imports --allow-subclassing-any
```

Finally, run all the tests locally like this:

```
pytest -vv --cov-report term-missing --cov-report=xml --cov=mirdata --black tests/ --
↪local
```

The *–local* flag skips tests that are built to run only on the remote testing environment.

To run one specific test file:

```
pytest tests/datasets/test_ikala.py
```

Finally, there is one local test you should run, which we can't easily run in our testing environment.

```
pytest -s tests/test_full_dataset.py --local --dataset dataset
```

Where `dataset` is the name of the module of the dataset you added. The `-s` tells pytest not to skip print statements, which is useful here for seeing the download progress bar when testing the download function.

This tests that your dataset downloads, validates, and loads properly for every track. This test takes a long time for some datasets, but it's important to ensure the integrity of the library.

The `--skip-download` flag can be added to `pytest` command to run the tests skipping the download. This will skip the downloading step. Note that this is just for convenience during debugging - the tests should eventually all pass without this flag.

Reducing the testing space usage

We are trying to keep the test resources folder size as small as possible, because it can get really heavy as new loaders are added. We kindly ask the contributors to **reduce the size of the testing data** if possible (e.g. trimming the audio tracks, keeping just two rows for csv files).

4. Submit your loader

Before you submit your loader make sure to:

1. Add your module to `docs/source/mirdata.rst` following an alphabetical order
2. Add your module to `docs/source/table.rst` following an alphabetical order as follows:

```
* - Dataset
  - Downloadable?
  - Annotation Types
  - Tracks
  - License
```

An example of this for the `Beatport EDM key` dataset:

```
* - Beatport EDM key
  - - audio:
    - annotations:
  - - global :ref:`key`
  - 1486
  - .. image:: https://licensebuttons.net/l/by-sa/3.0/88x31.png
       :target: https://creativecommons.org/licenses/by-sa/4.0
```

(you can check that this was done correctly by clicking on the readthedocs check when you open a PR). You can find license badges images and links here.

Pull Request template

When starting your PR please use the new_loader.md template, it will simplify the reviewing process and also help you make a complete PR. You can do that by adding `&template=new_loader.md` at the end of the url when you are creating the PR :

`...mir-dataset-loaders/mirdata/compare?expand=1` will become `...mir-dataset-loaders/mirdata/compare?expand=1&template=new_loader.md`.

Docs

Staged docs for every new PR are built, and you can look at them by clicking on the "readthedocs" test in a PR. To quickly troubleshoot any issues, you can build the docs locally by navigating to the `docs` folder, and running `make html` (note, you must have `sphinx` installed). Then open the generated `_build/source/index.html` file in your web browser to view.

Troubleshooting

If github shows a red `X` next to your latest commit, it means one of our checks is not passing. This could mean:

1. running `black` has failed – this means that your code is not formatted according to `black`'s code-style. To fix this, simply run the following from inside the top level folder of the repository:

```
black mirdata/ tests/
```

2. Your code does not pass `flake8` test.

```
flake8 mirdata --count --select=E9,F63,F7,F82 --show-source --statistics
```

3. Your code does not pass `mypy` test.

```
python -m mypy mirdata --ignore-missing-imports --allow-subclassing-any
```

4. the test coverage is too low – this means that there are too many new lines of code introduced that are not tested.
5. the docs build has failed – this means that one of the changes you made to the documentation has caused the build to fail. Check the formatting in your changes and make sure they are consistent.
6. the tests have failed – this means at least one of the tests is failing. Run the tests locally to make sure they are passing. If they are passing locally but failing in the check, open an *issue* and we can help debug.

2.9.3 Common non-standard cases

Not fully-downloadable datasets

Sometimes, parts of music datasets are not freely available due to e.g. copyright restrictions. In these cases, we aim to make sure that the version used in mirdata is the original one, and not a variant.

Before starting a PR, if a dataset **is not fully downloadable**:

1. Contact the mirdata team by opening an issue or PR so we can discuss how to proceed with the closed dataset.
2. Show that the version used to create the checksum is the "canonical" one, either by getting the version from the dataset creator, or by verifying equivalence with several other copies of the dataset.

Datasets needing extra dependencies

If a new dataset requires a library that is not included setup.py, please open an issue. In general, if the new library will be useful for many future datasets, we will add it as a dependency. If it is specific to one dataset, we will add it as an optional dependency.

To add an optional dependency, add the dataset name as a key in *extras_require* in setup.py, and list any additional dependencies. Additionally, mock the dependencies in docs/conf.py by adding it to the *autodoc_mock_imports* list.

When importing these optional dependencies in the dataset module, use a try/except clause and log instructions if the user hasn't installed the extra requirements.

For example, if a module called *example_dataset* requires a module called *asdf*, it should be imported as follows:

```
try:
    import asdf
except ImportError:
    logging.error(
        "In order to use example_dataset you must have asdf installed. "
        "Please reinstall mirdata using `pip install 'mirdata[example_dataset]'"
    )
    raise ImportError
```

Datasets with multiple versions

There are some datasets where the loading code is the same, but there are multiple versions of the data (e.g. updated annotations, or an additional set of tracks which follow the same paradigm). In this case, only one loader should be written, and multiple versions can be defined by creating additional indexes. Indexes follow the naming convention <datasetname>_index_<version>.json, thus a dataset with two versions simply has two index files. Different versions are tracked using the `INDEXES` variable:

```
INDEXES = {
    "default": "1.0",
    "test": "sample",
    "1.0": core.Index(filename="example_index_1.0.json"),
    "2.0": core.Index(filename="example_index_2.0.json"),
    "sample": core.Index(filename="example_index_sample.json")
}
```

By default, mirdata loads the version specified as `default` in `INDEXES` when running `mirdata.initialize('example')`, but a specific version can be loaded by running `mirdata.initialize('example', version='2.0')`.

Different indexes can refer to different subsets of the same larger dataset, or can reference completely different data. All data needed for all versions should be specified via keys in `REMOTES`, and by default, mirdata will download everything. If one version only needs a subset of the data in `REMOTES`, it can be specified using the `partial_download` argument of `core.Index`. For example, if `REMOTES` has the keys `['audio', 'v1-annotations', 'v2-annotations']`, the `INDEXES` dictionary could look like:

```
INDEXES = {
    "default": "1.0",
    "test": "1.0",
    "1.0": core.Index(filename="example_index_1.0.json", partial_download=['audio', 'v1-
↪annotations']),
    "2.0": core.Index(filename="example_index_2.0.json", partial_download=['audio', 'v2-
```

(continues on next page)

(continued from previous page)

```
↪annotations']),
}
```

Datasets with large indexes

Large indexes should be stored remotely, rather than checked in to the mirdata repository. mirdata has a zenodo community where larger indexes can be uploaded as "datasets".

When defining a remote index in `INDEXES`, simply also pass the arguments `url` and `checksum` to the `Index` class:

```
"1.0": core.Index(
    filename="example_index_1.0.json",  # the name of the index file
    url=<url>,  # the download link
    checksum=<checksum>,  # the md5 checksum
)
```

Remote indexes get downloaded along with the data when calling `.download()`, and are stored in `<data_home>/mirdata_indexes`.

2.9.4 Documentation

This documentation is in rst format. It is built using Sphinx and hosted on readthedocs. The API documentation is built using autodoc, which autogenerates documentation from the code's docstrings. We use the napoleon plugin for building docs in Google docstring style. See the next section for docstring conventions.

mirdata uses Google's Docstring formatting style. Here are some common examples.

Note: The small formatting details in these examples are important. Differences in new lines, indentation, and spacing make a difference in how the documentation is rendered. For example writing `Returns:` will render correctly, but `Returns` or `Returns :` will not.

Functions:

```
def add_to_list(list_of_numbers, scalar):
    """Add a scalar to every element of a list.
    You can write a continuation of the function description here on the next line.

    You can optionally write more about the function here. If you want to add an example
    of how this function can be used, you can do it like below.

    Example:
        .. code-block:: python

        foo = add_to_list([1, 2, 3], 2)

    Args:
        list_of_numbers (list): A short description that fits on one line.
        scalar (float):
            Description of the second parameter. If there is a lot to say you can
            overflow to a second line.
```

(continues on next page)

(continued from previous page)

```
    Returns:
        list: Description of the return. The type here is not in parentheses

    """
    return [x + scalar for x in list_of_numbers]
```

Functions with more than one return value:

```
def multiple_returns():
    """This function has no arguments, but more than one return value. Autodoc with␣
↪napoleon doesn't handle this well,
    and we use this formatting as a workaround.

    Returns:
        * int - the first return value
        * bool - the second return value

    """
    return 42, True
```

One-line docstrings

```
def some_function():
    """
    One line docstrings must be on their own separate line, or autodoc does not build␣
↪them properly
    """
    ...
```

Objects

```
"""Description of the class
overflowing to a second line if it's long

Some more details here

Args:
    foo (str): First argument to the __init__ method
    bar (int): Second argument to the __init__ method

Attributes:
    foobar (str): First track attribute
    barfoo (bool): Second track attribute

Cached Properties:
    foofoo (list): Cached properties are special mirdata attributes
    barbar (None): They are lazy loaded properties.
    barf (bool): Document them with this special header.

"""
```

2.9.5 Conventions

Opening files

Mirdata uses the smart_open library under the hood in order to support reading data from remote filesystems. If your loader needs to either call the python `open` command, or if it needs to use `os.path.exists`, you'll need to include the line

```
from smart_open import open
```

at the top of your dataset module and use `open` as you normally would. Sometimes dependency libraries accept file paths as input to certain functions and open the files internally - whenever possible mirdata avoids this, and passes in file-objects directly.

If you just need `os.path.exists`, you'll need to replace it with a try/except:

```
# original code that uses os.path.exists
file_path = "flululu.txt"
if not os.path.exists(file_path):
    raise FileNotFoundError(f"{file_path} not found, did you run .download?")

with open(file_path, "r") as fhandle:
    ...

# replacement code that is compatible with remote filesystems
try:
    with open(file_path, "r") as fhandle:
        ...
except FileNotFoundError:
    raise FileNotFoundError(f"{file_path} not found, did you run .download?")
```

Loading from files

We use the following libraries for loading data from files:

Format	library
audio (wav, mp3, ...)	librosa
midi	pretty_midi
json	json
csv	csv
jams	jams
yaml	pyyaml
hdf5 / h5	h5py

If a file format needed for a dataset is not included in this list, please see *this section*

Track Attributes

If the dataset has an official e.g. train/test split, use the reserved attribute *Track.split*, or *MultiTrack.split* which will enable some dataset-level helper functions like *dataset.get_track_splits*. If there is no official split, do not use this attribute.

Custom track attributes should be global, track-level data. For some datasets, there is a separate, dataset-level metadata file with track-level metadata, e.g. as a csv. When a single file is needed for more than one track, we recommend using writing a `_metadata` cached property (which returns a dictionary, either keyed by track_id or freeform) in the Dataset class (see the dataset module example code above). When this is specified, it will populate a track's hidden `_track_metadata` field, which can be accessed from the Track class.

For example, if `_metadata` returns a dictionary of the form:

```
{
    'track1': {
        'artist': 'A',
        'genre': 'Z'
    },
    'track2': {
        'artist': 'B',
        'genre': 'Y'
    }
}
```

the `_track metadata` for `track_id=track2` will be:

```
{
    'artist': 'B',
    'genre': 'Y'
}
```

Missing Data

If a Track has a property, for example a type of annotation, that is present for some tracks and not others, the property should be set to `None` when it isn't available.

The index should only contain key-values for files that exist.

2.9.6 Custom Decorators

cached_property

This is used primarily for Track classes.

This decorator causes an Object's function to behave like an attribute (aka, like the `@property` decorator), but caches the value in memory after it is first accessed. This is used for data which is relatively large and loaded from files.

docstring_inherit

This decorator is used for children of the Dataset class, and copies the Attributes from the parent class to the docstring of the child. This gives us clear and complete docs without a lot of copy-paste.

coerce_to_bytes_io/coerce_to_string_io

These are two decorators used to simplify the loading of various `Track` members in addition to giving users the ability to use file streams instead of paths in case the data is in a remote location e.g. GCS. The decorators modify the function to:

- Return `None` if `None` if passed in.
- Open a file if a string path is passed in either `'w'` mode for `string_io` or `wb` for `bytes_io` and pass the file handle to the decorated function.
- Pass the file handle to the decorated function if a file-like object is passed.

This cannot be used if the function to be decorated takes multiple arguments. `coerce_to_bytes_io` should not be used if trying to load an mp3 with librosa as libsndfile does not support `mp3` yet and `audioread` expects a path.

2.10 FAQ

2.10.1 How do I add a new loader?

Take a look at our *Contributing* docs!

2.10.2 How do I get access to a dataset if the download function says it's not available?

We don't distribute data ourselves, so unfortunately it is up to you to find the data yourself. We strongly encourage you to favor datasets which are currently available.

2.10.3 Can you send me the data for a dataset which is not available?

No, we do not host or distribute datasets.

2.10.4 How do I request a new dataset?

Open an issue and tag it with the "New Loader" label.

2.10.5 What do I do if my data fails validation?

Very often, data fails vaildation because of how the files are named or how the folder is structured. If this is the case, try renaming/reorganizing your data to match what mirdata expects. If your data fails validation because of the checksums, this means that you are using data which is different from what most people are using, and you should try to get the more common dataset version, for example by using the data loader's download function.

2.10.6 How do you choose the data that is used to create the checksums?

Whenever possible, the data downloaded using :code:`.download()` is the same data used to create the checksums. If this isn't possible, we did our best to get the data from the original source (the dataset creator) in order to create the checksum. If this is again not possible, we found as many versions of the data as we could from different users of the dataset, computed checksums on all of them and used the version which was the most common amongst them.

2.10.7 Does mirdata provide data loaders for pytorch/Tensorflow?

For now, no. Music datasets are very widely varied in their annotation types and supported tasks. To make a data loader, there would need to be "standard" ways to encode the desired inputs/outputs - unfortunately this is not universal for most datasets and usages. Still, this library provides the necessary first step for building data loaders and it is easy to build data loaders on top of this. For more information, see *Using mirdata with tensorflow*.

2.10.8 A download link is broken for a loader's :code:`.download()` function. What do I do?

Please open an issue and tag it with the "broken link" label.

2.10.9 Why the name, mirdata?

mirdata = mir + data. MIR is an acronym for Music Information Retrieval, and the library was built for working with data.

2.10.10 If I find a mistake in an annotation, should I fix it in the loader?

No. All datasets have "mistakes", and we do not want to create another version of each dataset ourselves. The loaders should load the data as released. After that, it's up to the user what they want to do with it.

2.10.11 Does mirdata support data which lives off-disk?

Yes! We use the smart_open library, which supports non-local filesystems such as GCS and AWS. See *Accessing data on non-local filesystems* for details.

BIBLIOGRAPHY

[giantsteps_tempo_cit_1] Peter Knees, Ángel Faraldo, Perfecto Herrera, Richard Vogl, Sebastian Böck, Florian Hörschläger, Mickael Le Goff: "Two data sets for tempo estimation and key detection in electronic dance music annotated from user corrections", Proc. of the 16th Conference of the International Society for Music Information Retrieval (ISMIR'15), Oct. 2015, Malaga, Spain.

[giantsteps_tempo_cit_2] Hendrik Schreiber, Meinard Müller: "A Crowdsourced Experiment for Tempo Estimation of Electronic Dance Music", Proc. of the 19th Conference of the International Society for Music Information Retrieval (ISMIR'18), Sept. 2018, Paris, France.

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

C

D

E

F

G

I

J

K

L

M

U

V